CS 106A, Lecture 18
Practice with 1D and 2D Arrays
Plan for Today

• Recap: 2D Arrays and Images
• Practice: Shrink
• Practice: Cryptogram
• Practice: Tic-Tac-Toe
Plan for Today

• Recap: 2D Arrays and Images
• Practice: Shrink
• Practice: Cryptogram
• Practice: Tic-Tac-Toe
The Matrix
2D Arrays ("Matrices")

WELCOME...... TO THE MATRIX!!!!!!
2D Arrays

type[][] name = new type[rows][columns];

int[][] a = new int[3][5];

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
<td>a[0][4]</td>
</tr>
<tr>
<td>1</td>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
<td>a[1][4]</td>
</tr>
</tbody>
</table>
int[][] a = new int[3][4];
int[] firstRow = a[0];
Summary: 2D Arrays

- Make a new 2D array

\[
\text{type[][] name} = \text{new type[rows][columns]};
\]

- Get and set values using bracket notation

\[
\text{name[row][col]} \quad \text{// get elem at row,col}
\]
\[
\text{name[row][col]} = \text{value}; \quad \text{// set elem at row,col}
\]

- Get the number of rows and columns

\[
\text{arr.length} \quad \text{// # rows}
\]
\[
\text{arr[0].length} \quad \text{// # columns}
\]

- Iterate over a 2D array using a double for-loop

\[
\text{for (int row = 0; row < arr.length; row++)} \{
\text{for (int col = 0; col < arr[0].length; col++)} \{
\text{// do something with arr[row][col];}
\text{}}
\text{}}
\]
Limitations of 2D Arrays

• Unlike 1D arrays, you cannot compare 2D arrays with `Arrays.equals`. You must use `Arrays.deepEquals`.

```java
int[][][] a1 = ...;
int[][][] a2 = ...;
if (Arrays.deepEquals(a1, a2)) {  ...  }
```

• A 2D array does not know how to print itself:

```java
int[][][] a = new int[rows][cols];
println(a);  // [[I@8cf420
println(Arrays.toString(a));  // [[I@6b3f44,[I@32c2a8]...

// [[0, 1, 2, 3, 4], [1, 2, ... println(Arrays.deepToString(a));
```
Images are just grids (2D arrays) of pixels! Pixels are just integer values that have red, green, and blue components (each between 0 and 255).
Example: Pointillism

Pointillism is an art style where many small dots of color are combined to make a larger image.
Red, Green and Blue in one int?

Images encode the R, G, and B values (between 0 and 255) of a pixel into a single integer. You can convert between this pixel value and the individual RGB values.

```java
int[][][] pixels = image.getPixelArray();
int px = pixels[0][0];
int red = GImage.getRed(px);
int green = GImage.getGreen(px);
int blue = GImage.getBlue(px);
```
Images *encode* the R, G, and B values (between 0 and 255) of a pixel into a single integer. You can convert between this **pixel value** and the individual **RGB values**.

You can also create pixels with your own RGB values.

```java
int r = ...;
int g = ...;
int b = ...;
int pixel = GImage.createRGBPixel(r, g, b);
```
We can get a GImage as a 2D array of pixels, and modify it any way we want. Then, we can create a new GImage with the modified pixels.

```java
GImage img = new GImage("res/snowman.jpg");
int[][][] pixels = img.getPixelArray();
...
// (modify pixels)
img.setPixelArray(pixels);  // update image

// or make a new GImage
GImage newImg = new GImage(pixels);
```
Modifying Image Pixels

- There are many cool image algorithms based around modifying individual pixels in an image: grayscale, brighten, normalize, remove red-eye...
GImage img = new GImage("res/daisy.jpg");

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>img</strong>.getPixelArray()</td>
<td>returns pixels as 2D array of ints, where each int in the array contains all 3 of Red, Green, and Blue merged into a single integer</td>
</tr>
<tr>
<td><strong>img</strong>.setPixelArray(\textit{array});</td>
<td>updates pixels using the given 2D array of ints</td>
</tr>
<tr>
<td>GImage.createRGBPixel(\textit{r, g, b})</td>
<td>returns an int that merges the given amounts of red, green and blue (each 0-255)</td>
</tr>
<tr>
<td>GImage.getRed(\textit{px})</td>
<td>returns the redness, greenness, or blueness of the given pixel as an integer from 0-255</td>
</tr>
<tr>
<td>GImage.getGreen(\textit{px})</td>
<td></td>
</tr>
<tr>
<td>GImage.getBlue(\textit{px})</td>
<td></td>
</tr>
</tbody>
</table>
Recap: Modifying Pixels

- **Extract** pixel RGB colors with `GImage.getRed/Blue/Green`.

  ```java
  int red   = GImage.getRed(pixels[0][0]);  // 0-255
  int green = GImage.getGreen(pixels[0][0]); // 0-255
  int blue  = GImage.getBlue(pixels[0][0]);  // 0-255
  ```

- **Modify** the color components for a given pixel.

  ```java
  red = 0;  // remove redness
  ```

- **Combine** the RGB back together into a single int.

  ```java
  pixels[0][0] = GImage.createRGBPixel(red, green, blue);
  ```

- **Update** the image with your modified pixels when finished.

  ```java
  image.setPixelArray(pixels);
  ```
Plan for Today

- Recap: 2D Arrays and Images
- Practice: Shrink
- Practice: Cryptogram
- Practice: Tic-Tac-Toe
Shrink

Let’s write a program that can *shrink* an image to \( \frac{1}{2} \) its original size.
Shrink

Given a pixel \((x, y)\) in our smaller image, how do we know which pixel in our larger image should go there?
Given a pixel \((x, y)\) in our smaller image, how do we know which pixel in our larger image should go there?
Given a pixel \((x, y)\) in our smaller image, how do we know which pixel in our larger image should go there?
Shrink

Given a pixel \((x, y)\) in our smaller image, how do we know which pixel in our larger image should go there?
int[][][] pixels = image.getPixelArray();
int[][][] result =
    new int[pixels.length/2][pixels[0].length/2];

for (int r = 0; r < result.length; r++) {
    for (int c = 0; c < result[0].length; c++) {
        result[r][c] = pixels[r*2][c*2];
    }
}

image.setPixelArray(result);
Shrink

```java
int[][][] pixels = image.getPixelArray();
int[][][] result =
    new int[pixels.length/2][pixels[0].length/2];

for (int r = 0; r < result.length; r++) {
    for (int c = 0; c < result[0].length; c++) {
        result[r][c] = pixels[r*2][c*2];
    }
}

image.setPixelArray(result);
```
int[][][] pixels = image.getPixelArray();
int[][][] result =
    new int[pixels.length/2][pixels[0].length/2];

for (int r = 0; r < result.length; r++) {
    for (int c = 0; c < result[0].length; c++) {
        result[r][c] = pixels[r*2][c*2];
    }
}

image.setPixelArray(result);
int[][][] pixels = image.getPixelArray();
int[][][] result =
    new int[pixels.length/2][pixels[0].length/2];

for (int r = 0; r < result.length; r++) {
    for (int c = 0; c < result[0].length; c++) {
        result[r][c] = pixels[r*2][c*2];
    }
}

image.setPixelArray(result);
Shrink

```java
int[][][] pixels = image.getPixelArray();
int[][][] result =
    new int[pixels.length/2][pixels[0].length/2];

for (int r = 0; r < result.length; r++) {
    for (int c = 0; c < result[0].length; c++) {
        result[r][c] = pixels[r*2][c*2];
    }
}

image.setPixelArray(result);
```
Shrink

```java
int[][][] pixels = image.getPixelArray();
int[][][] result =
    new int[pixels.length/2][pixels[0].length/2];

for (int r = 0; r < result.length; r++) {
    for (int c = 0; c < result[0].length; c++) {
        result[r][c] = pixels[r*2][c*2];
    }
}

image.setPixelArray(result);
```
Plan for Today

• Recap: 2D Arrays and Images
• Practice: Shrink
• Practice: Cryptogram
• Practice: Tic-Tac-Toe
A **cryptogram** is a puzzle in which a message is encoded by replacing each letter in the original text with some other letter. Your job in solving a cryptogram is figuring out this substitution pattern.

A common technique is assuming the most common letters in the coded message correspond to the most common letters in English.

Edgar Allan Poe (1809-1849)
Letter Frequency

<table>
<thead>
<tr>
<th>LET</th>
<th>COUNT</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>445.2 B</td>
<td>12.49%</td>
</tr>
<tr>
<td>T</td>
<td>330.5 B</td>
<td>9.28%</td>
</tr>
<tr>
<td>A</td>
<td>286.5 B</td>
<td>8.04%</td>
</tr>
<tr>
<td>O</td>
<td>272.3 B</td>
<td>7.64%</td>
</tr>
<tr>
<td>I</td>
<td>269.7 B</td>
<td>7.57%</td>
</tr>
<tr>
<td>N</td>
<td>257.8 B</td>
<td>7.23%</td>
</tr>
<tr>
<td>S</td>
<td>232.1 B</td>
<td>6.51%</td>
</tr>
<tr>
<td>R</td>
<td>223.8 B</td>
<td>6.28%</td>
</tr>
<tr>
<td>H</td>
<td>180.1 B</td>
<td>5.05%</td>
</tr>
<tr>
<td>L</td>
<td>145.0 B</td>
<td>4.07%</td>
</tr>
<tr>
<td>D</td>
<td>136.0 B</td>
<td>3.82%</td>
</tr>
<tr>
<td>C</td>
<td>119.2 B</td>
<td>3.34%</td>
</tr>
<tr>
<td>U</td>
<td>97.3 B</td>
<td>2.73%</td>
</tr>
<tr>
<td>M</td>
<td>89.5 B</td>
<td>2.51%</td>
</tr>
<tr>
<td>F</td>
<td>85.6 B</td>
<td>2.40%</td>
</tr>
<tr>
<td>P</td>
<td>76.1 B</td>
<td>2.14%</td>
</tr>
<tr>
<td>G</td>
<td>66.6 B</td>
<td>1.87%</td>
</tr>
<tr>
<td>W</td>
<td>59.7 B</td>
<td>1.68%</td>
</tr>
<tr>
<td>Y</td>
<td>59.3 B</td>
<td>1.66%</td>
</tr>
<tr>
<td>B</td>
<td>52.9 B</td>
<td>1.48%</td>
</tr>
<tr>
<td>V</td>
<td>37.5 B</td>
<td>1.05%</td>
</tr>
<tr>
<td>K</td>
<td>19.3 B</td>
<td>0.54%</td>
</tr>
<tr>
<td>X</td>
<td>8.4 B</td>
<td>0.23%</td>
</tr>
<tr>
<td>J</td>
<td>5.7 B</td>
<td>0.16%</td>
</tr>
<tr>
<td>Q</td>
<td>4.3 B</td>
<td>0.12%</td>
</tr>
<tr>
<td>Z</td>
<td>3.2 B</td>
<td>0.09%</td>
</tr>
</tbody>
</table>

By Peter Norvig
Head of Google Research
Former CS221 Instructor

Based on 3.8 trillion letters
Poe’s Cryptographic Puzzle

5345;305;6;48264;806;4888
85;15;883885;46;8896;
8;485;52:885;4956254
8;4069285;68;4885;4855
1;4885;4455;81(9;48;88;4
?3448)4;161;188;?;
Poe’s Cryptographic Puzzle

53‡ ‡ ‡ 3 0 5 ) 6 * ; 4 8 2 6 ) 4 ‡ • ) 4 ‡ ) ; 8 0 6 * ; 4 8 ‡ 8 ¶ 6 0 ) ) 8 5 ; 1 ‡ ( ; : ‡ * 8 ‡ 8 3 ( 8 8 ) 5 * ‡ ; 4 6 ( ; 8 8 * 9 6 * ? ; 8 ) * ‡ ( ; 4 8 5 ) ; 5 * ‡ 2 : ‡ ( ; 4 9 5 6 * 2 ( 5 * - 4 ) 8 ¶ 8 * ; 4 0 6 9 2 8 5 ) ; ) 6 ‡ 8 ) 4 ‡ ‡ ; 1 ( ‡ 9 ; 4 8 0 8 1 ; 8 : 8 ‡ 1 ; 4 8 ‡ 8 5 ; 4 ) 4 8 5 ‡ 5 2 8 8 0 6 * 8 1 ( ‡ 9 ; 4 8 ; ( 8 8 ; 4 ( ‡ ? 3 4 ; 4 8 ) 4 ‡ ; 1 6 1 ; : 1 8 8 ; ‡ ? ;

AGOODGLASSINTHEBISHOPSHOTELINTHEDEVL ISSEATFORTYONEDEGREESSANDTHIRTEENMIN UTESNORTHEASTANDBYNORTHMAINBRANCHSEV ENTHLIMBEASTSIDESHOOTFROMTHELEFTYEOTO FTHEDEATHSHEHEADABEELINEFROMTHTREETHR OUGHTHESHOTFIFTYFEETOUT

8 33
; 26
4 19
‡ 16
) 16
* 13
5 12
6 11
( 10
† 8
1 8
0 6
9 5
2 5
: 4
3 4
? 3
¶ 2
- 1
• 1
Idea: Array of Counters

• For problems like this, where we want to keep count of many things, a *frequency table* (or *tally array*) can be a clever solution.
  – *Idea*: The element at index $i$ will store a counter for the character value ‘A’ + $i$.

– example: count of letter frequency for ”FIDDLE”

<table>
<thead>
<tr>
<th>letter</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>J</th>
<th>L</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>...</td>
</tr>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

36
Plan for Today

• Recap: 2D Arrays and Images
• Practice: Shrink
• Practice: Cryptogram
• Practice: Tic-Tac-Toe
Let’s use 2D arrays to create a ConsoleProgram version of Tic-Tac-Toe.
Recap:

• Recap: 2D Arrays and Images
• Practice: Shrink
• Practice: Cryptogram
• Practice: Tic-Tac-Toe

Next Time: More data structures