Alice Diakova
Professor Kender
Visual Interfaces of Computers
4/10/23

Assignment 3

Step 1:

Code:

```
# important imports

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import math
from scipy import ndimage

# STEP 1: RAW DATA

# Reading Table.txt to create dictionary mapping building numbers to names
buildings = dict()
f = open("Desktop/vis_int_hw3/Table.txt", "r")
#print(f.read())
for line in f:
    split_line = line.split()
    buildings[int(split_line[0])] = split_line[1]
f.close()
print(buildings)
```

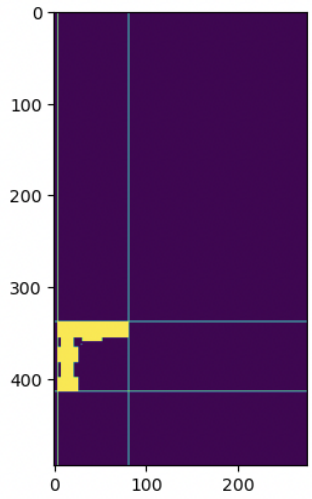
```
{9: 'Pupin', 19: 'SchapiroCEPSR', 28: 'Mudd&EngTerrace&Fairchild&CS', 38: 'NorthwestCorner', 47: 'Uris', 57: 'Scherme
rhorn', 66: 'Chandler&Havemeyer', 76: 'OldComputerCenter', 85: 'Avery', 94: 'Fayerweather', 104: 'Mathematics', 113:
'LowLibrary', 123: 'StPaulChapel', 132: 'EarlHall', 142: 'Lewisohn', 151: 'Philosophy', 161: 'Buell', 170: 'AlmaMate
r', 179: 'Dodge', 189: 'Kent', 198: 'CollegeWalk', 208: 'Journalism&Furnald', 217: 'Hamilton&Hartley&Wallach&JohnJa
y', 236: 'Lerner', 246: 'ButlerLibrary', 255: 'Carman'}
```

```
# Creating Numpy Array from 'Labeled.pgm' file
labeled_file = Image.open('Desktop/vis_int_hw3/Labeled.pgm')
labeled_arr = np.asarray(labeled_file)

# <class 'numpy.ndarray'>
print(type(labeled_arr))

#  shape
print(labeled_arr.shape)

print(labeled_arr)
```

```
<class 'numpy.ndarray'>
(495, 275)
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

```python
# NOTE: The center of mass calculation performed in this function using the ndimage was found
# from the following source:
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.center_of_mass.html


# Function that creates a 2D numpy binary array where pixels with a 1 indicate the location of the
# target building while 0's indicate all other pixels (including space between buildings and other buildings)
# This function also returns valuable information about the building including area in pixels, com, and mbr info
def building_layer_area_com_mbr(target_num, labeled_arr):
    # target_num is int representing target building number
    # labeled_arr is np array version of Labeled.pgm file
    # returns tuple of length four containing the following info:
        # 1: binary np array with same size as labeled_arr with 1's in locations where target building
            # pixels are located and 0's in all other locations
        # 2: total area in pixels
        # 3: tuple of size 2 containing (x,y) coords of target building Center of Mass
        # 4: tuple of size 3 containing (mbr upper left coordinate (x,y) tuple,
        #                                mbr lower right coordinate (x,y) tuple, length of diagonal)
    layer = np.copy(labeled_arr)
    area = 0
    height = labeled_arr.shape[0]
    width = labeled_arr.shape[1]

    leftmost_x = width-1
    rightmost_x = 0
    lowermost_y = 0
    uppermost_y = height-1

    for row in range(height):
        for col in range(width):
            if labeled_arr[row,col] == target_num:
                layer[row,col] = 1

                area += 1

                rightmost_x = max(col, rightmost_x)
                leftmost_x = min(col, leftmost_x)
                uppermost_y = min(row, uppermost_y)
                lowermost_y = max(row, lowermost_y)

            else:
                layer[row,col] = 0

    nd_com = ndimage.center_of_mass(layer) # returns (row, col) not (x,y)
    x = int(nd_com[1])
    y = int(nd_com[0])
    #x = ((rightmost_x - leftmost_x)/2.0) + leftmost_x
    #y = ((lowermost_y - uppermost_y)/2.0) + uppermost_y
    com = (x,y)

    mbr_upperleft = (leftmost_x, uppermost_y)
    mbr_lowerright = (rightmost_x, lowermost_y)
    diag = round(math.dist(mbr_upperleft, mbr_lowerright), 1)
    mbr = (mbr_upperleft, mbr_lowerright, diag)

    return (layer, area, com, mbr)

# code that tests building_layer_area_com_mbr function by printing area and ploting the test building layer including
    # COM coord point and lines representing the MBR

test_target_num = 208

test = building_layer_area_com_mbr(test_target_num, labeled_arr)
test_layer = test[0]
test_area = test[1]
test_com = test[2]
test_mbr = test[3]
test_mbr_upperleftcorner = test_mbr[0]
test_mbr_lowerrightcorner = test_mbr[1]
test_mbr_diagonal = test_mbr[2]

print("Area: ", test_area, " | COM: ", test_com, " | MBR: ", test_mbr)

if test_layer[int(test_com[1]), int(test_com[0])] == 1:
    test_layer[int(test_com[1]), int(test_com[0])] = 0 # adding com to display
else:
    test_layer[int(test_com[1]), int(test_com[0])] = 1
for row in range(len(test_layer)):
    for col in range(len(test_layer[0])):
        if row == test_mbr_upperleftcorner[1] or row == test_mbr_lowerrightcorner[1]:
            test_layer[row, col] = 1
        if col == test_mbr_upperleftcorner[0] or col == test_mbr_lowerrightcorner[0]:
            test_layer[row, col] = 1

plt.imshow(test_layer)
```

```
Area:  2615  | COM:  (30, 363)  | MBR:  ((4, 338), (81, 414), 108.2)
```




```
# creating dictionary containing all data of all buildings
raw_data = dict()
# raw_data keys are building numbers
# raw_data values are lists of size 8 containing the following info:
    # 1: str -> building name
    # 2: np array -> building layer
    # 3: tuple -> (x,y) coord of center of mass
    # 4: int -> total area in pixels
    # 5: tuple -> (x,y) coord of upper left corner of MBR
    # 6: tuple -> (x,y) coord of lower right corner of MBR
    # 7: float -> len of MBR diagonal
    # 8: list of strings -> all buildings who's MBR intersects with the MBR of the target building

for building in buildings.items():
    building_num = building[0]
    building_name = building[1]

    layer, area, com, mbr = building_layer_area_com_mbr(building_num, labeled_arr)

    raw_data[building_num] = [building_name, layer, com, area, mbr[0], mbr[1], mbr[2], []]

# function taking in raw data and filling in the intersection list

def determine_intersections(raw_data, input_building):
    # input: raw_data, building number
    # task: determines which buildings intersect with the mbr of the input building and updates raw_data[input_building
    # output: None

    build_info = raw_data[input_building]
    l1 = build_info[4]
    r1 = build_info[5]

    intersect = list()

    for key,val in raw_data.items():
        if key == input_building:
            continue

        build2_info = val
        l2 = build2_info[4]
        r2 = build2_info[5]

        # If one rectangle is on left side of other
        if l1[0] >= r2[0] or l2[0] >= r1[0]:
            continue

        # If one rectangle is above other
        if r1[1] <= l2[1] or r2[1] <= l1[1]:
            continue

        print('intersection')
        print(l1, l2)
        print(r1, r2)
        intersect.append(build2_info[0])

    #raw_data[input_building][7] = intersect

    return intersect
```

```
# updating raw_data by filling in the intersection list using the above determine_intersections function
for key, val in raw_data.items():
    raw_data[key][7] = determine_intersections(raw_data, key)

# testing intersection function by seeing what buildings' mbr's intersect with the mbr of building 28:
print(raw_data[28][7])
```

```
intersection
(166, 3) (110, 48)
(272, 86) (175, 147)
intersection
(166, 3) (181, 77)
(272, 86) (273, 147)
intersection
(110, 48) (166, 3)
(175, 147) (272, 86)
intersection
(181, 77) (166, 3)
(273, 147) (272, 86)
['Uris', 'Schermerhorn']
```

```
# Concluding section of STEP 1:
# printing, for each building, a table of building raw image properties including:
    # building name and number
    # (x,y) coords of the center of mass and the area in pixels
    # building min bounding rectangle info: (x,y) coords of upper left and lower right corners, diagonal length
    # any other buildings whos MBR intersect this building's MBR

# Print the names of the columns.
print("{:<3} {:<32} {:<10} {:<6} {:<10} {:<13} {:<6} {:<10}"
      .format('NUM', 'NAME', 'COM', 'AREA', 'MBR UL', 'MBR LR', 'DIAG', 'OVERLAP'))

raw_data[building_num] = [building_name, layer, com, area, mbr[0], mbr[1], mbr[2], []]

# print each data item.
for key, value in raw_data.items():
    name, layer, com, area, ul, lr, diag, overlap = value
    #if overlap == []:
    #    overlap = "N/A"
    #print("{:<3} {:<32} {:<10} {:<6} {:<10} {:<13} {:<5} {:<10}"
          #.format(str(key), name, str(com), str(area), str(ul), str(lr), str(diag), str(overlap)))

    print("{:<3} {:<32} {:<10} {:<6} {:<10} {:<13} {:<6}"
          .format(str(key), name, str(com), str(area), str(ul), str(lr), str(diag)), end=" ")
    if overlap == []:
        overlap = "N/A"
    else:
        for x in overlap:
            print(x, end=" ")
    print()
```

NUM	NAME	COM	AREA	MBR UL	MBR LR	DIAG	OVERLAP
9	Pupin	(76, 14)	1640	(39, 3)	(115, 27)	79.7	
19	SchapiroCEPSR	(143, 20)	1435	(123, 3)	(163, 37)	52.5	
28	Mudd&EngTerrace&Fairchild&CS	(223, 35)	5831	(166, 3)	(272, 86)	134.6	Uris Schermerhorn
38	NorthwestCorner	(16, 40)	1998	(3, 4)	(29, 77)	77.5	
47	Uris	(142, 99)	5753	(110, 48)	(175, 147)	118.4	Mudd&EngTerrace&Fairchild&CS
57	Schermerhorn	(233, 120)	3911	(181, 77)	(273, 147)	115.6	Mudd&EngTerrace&Fairchild&CS
66	Chandler&Havemeyer	(37, 119)	3613	(3, 81)	(80, 147)	101.4	
76	OldComputerCenter	(96, 136)	322	(90, 125)	(103, 147)	25.6	
85	Avery	(204, 175)	1164	(191, 151)	(215, 201)	55.5	
94	Fayerweather	(259, 176)	1182	(247, 151)	(272, 201)	55.9	
104	Mathematics	(17, 182)	1191	(3, 158)	(31, 206)	55.6	
113	LowLibrary	(135, 221)	3898	(101, 187)	(169, 256)	96.9	
123	StPaulChapel	(226, 222)	1087	(201, 210)	(251, 234)	55.5	
132	EarlHall	(49, 221)	759	(31, 211)	(68, 233)	43.0	
142	Lewisohn	(17, 259)	1307	(3, 233)	(31, 285)	59.1	
151	Philosophy	(258, 263)	1085	(245, 240)	(272, 286)	53.3	
161	Buell	(208, 253)	340	(196, 246)	(220, 261)	28.3	
170	AlmaMater	(136, 276)	225	(129, 269)	(143, 283)	19.8	
179	Dodge	(41, 301)	1590	(3, 289)	(80, 311)	80.1	
189	Kent	(233, 300)	1470	(194, 290)	(272, 310)	80.5	
198	CollegeWalk	(137, 322)	4950	(0, 314)	(274, 331)	274.5	
208	Journalism&Furnald	(30, 363)	2615	(4, 338)	(81, 414)	108.2	
217	Hamilton&Hartley&Wallach&JohnJay	(240, 416)	5855	(191, 338)	(270, 490)	171.3	
236	Lerner	(38, 446)	2940	(4, 426)	(73, 467)	80.3	
246	ButlerLibrary	(132, 460)	5282	(85, 431)	(179, 490)	111.0	
255	Carman	(38, 479)	1540	(4, 469)	(73, 490)	72.1	

Discussion:

Displaying building number and name was relatively straightforward after distilling the Table.txt file into a dictionary relating the two. Next, to determine the center of mass I went through each building, creating a layer for each independently. By this I mean within a building's layer the only elements that have the value of 1 are in the locations of the pixels that belong to that building and that building alone- all other elements have the value zero. This layer was helpful as I inserted it into the ndimage center_of_mass method to determine the center of mass coordinates of each of these uniformly-massed rectangular shapes. While I was creating each layer I counted up the total number of pixels within each building to determine their areas. Additionally, within the layer loop that iterates through each pixel in the entire image I saved the left and upper most pixel of a building as well as the right lower most point and updated these points if I found a more extreme pixel. These values became my MBR upper left and lower right coordinates. I used these points to determine the diagonal length of the MBR for each building using the math.dist function. Finally, I created a function that iterates through each building and uses the upper left and lower right coordinates of each building to determine which buildings' MBR's overlap with one another. Within the output, I observed that each of the outputs seem reasonable with respect to the building they describe, thus I believe my design was successful for this step.

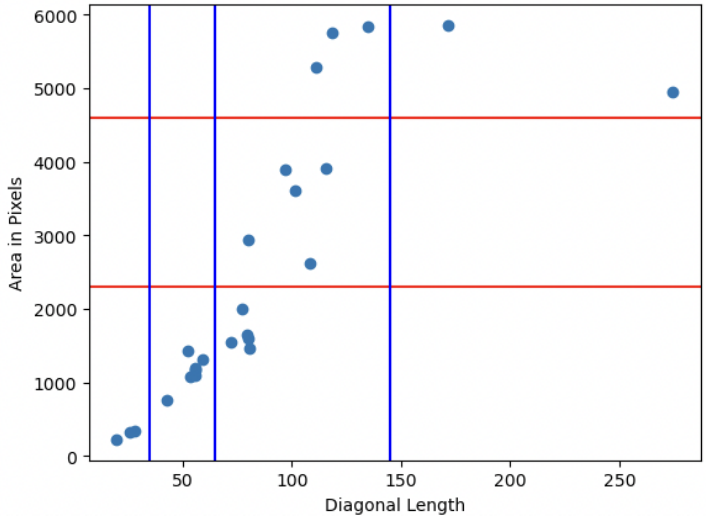
Step 2:

Code:

```
# SIZE
# the following few code blocks are used to determine the magic numbers to use to classify building size

# in this block I display area, diagonal length, name tuples sorted from largest area to least
size_list = list()
for key, val in raw_data.items():
    name, layer, com, area, ul, lr, diag, overlap = val
    size_list.append([area, diag, name])

size_list.sort()
size_list.reverse()

print('Area', 'Diag ', 'Name')
points = list()
for x in size_list:
    print(x[0], x[1],x[2])
    points.append([x[1], x[0]])
np_points = np.array(points)
```

```
Area Diag  Name
5855 171.3 Hamilton&Hartley&Wallach&JohnJay
5831 134.6 Mudd&EngTerrace&Fairchild&CS
5753 118.4 Uris
5282 111.0 ButlerLibrary
4950 274.5 CollegeWalk
3911 115.6 Schermerhorn
3898 96.9 LowLibrary
3613 101.4 Chandler&Havemeyer
2940 80.3 Lerner
2615 108.2 Journalism&Furnald
1998 77.5 NorthwestCorner
1640 79.7 Pupin
1590 80.1 Dodge
1540 72.1 Carman
1470 80.5 Kent
1435 52.5 SchapiroCEPSR
1307 59.1 Lewisohn
1191 55.6 Mathematics
1182 55.9 Fayerweather
1164 55.5 Avery
1087 55.5 StPaulChapel
1085 53.3 Philosophy
759 43.0 EarlHall
340 28.3 Buell
322 25.6 OldComputerCenter
225 19.8 AlmaMater
```

```
# size grouping determination graph block: in this block I create a scatterplot of the relationship
# between area and diagonal length, then I created lines that appear to seperate these points into
# sections that seem to be grouped together. I will label the points in each filled section with a unique label

plt.scatter(np_points[:,0], np_points[:,1])
plt.axhline(y = 4600, color = 'r', linestyle = '-')
plt.axhline(y = 2300, color = 'r', linestyle = '-')
plt.axvline(x = 35, color = 'b', label = 'axvline - full height')
plt.axvline(x = 65, color = 'b', label = 'axvline - full height')
plt.axvline(x = 145, color = 'b', label = 'axvline - full height')
plt.xlabel("Diagonal Length")
plt.ylabel("Area in Pixels")
plt.show()
```



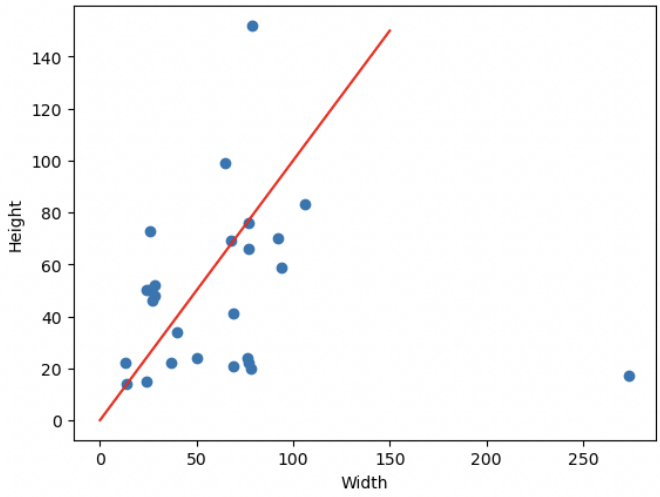
```python
# the following two functions are used to determine whether the inputted building is
# the largest or the smallest or neither

def isSmallest(raw_data, building_num):
    area = raw_data[building_num][3]
    diag = raw_data[building_num][6]
    input_combo = (area, diag)

    for key, val in raw_data.items():
        curr_combo = (val[3], val[6])
        if curr_combo < input_combo:
            return False

    return True

def isLargest(raw_data, building_num):
    area = raw_data[building_num][3]
    diag = raw_data[building_num][6]
    input_combo = (area, diag)

    for key, val in raw_data.items():
        curr_combo = (val[3], val[6])
        if curr_combo > input_combo:
            return False

    return True

# the following 3 labels derived from the graph above will be used to identify
# sizes distinctly in the function below.
    # Minuscule: the group with the smallest area and diagonal length will
    # Tiny: the group to the right (on the graph) of the Minuscule group
    # Small: the group to the right of the Tiny group
    # Medium: the middle group on the graph
    # Large: the upper middle group on the graph
    # Gigantic: the upper right group containing the points with the largest area and diagonal length

def size_(raw_data, building_num):
    # input: raw_data and building number
    # task: uses magic numbers derived from the graph above to label the size of the inputted building
    # output: string labeling the size of the input building

    area = raw_data[building_num][3]
    diag = raw_data[building_num][6]

#     if area < 2300:
#         return 'Small'

#     if area < 4600:
#         return 'Medium'

#     return 'Large'

    if isSmallest(raw_data, building_num):
        return 'Smallest'

    if isLargest(raw_data, building_num):
        return 'Largest'

    # Minuscule:
    if diag < 35 and area < 2300:
        return 'Minuscule'

    # Tiny
    if diag < 65 and area < 2300:
        return 'Tiny'

    # Small
    if diag < 145 and area < 2300:
        return 'Small'

    # Medium
    if diag < 145 and area < 4600:
        return 'Medium'

    # Large
    if diag < 145:
        return 'Large'

    # Gigantic
    return 'Gigantic'

# testing the above size functions:
for key, val in raw_data.items():
    print(val[0], size_(raw_data, key))
```

```
Pupin Small
SchapiroCEPSR Tiny
Mudd&EngTerrace&Fairchild&CS Large
NorthwestCorner Small
Uris Large
Schermerhorn Medium
Chandler&Havemeyer Medium
OldComputerCenter Minuscule
Avery Tiny
Fayerweather Tiny
Mathematics Tiny
LowLibrary Medium
StPaulChapel Tiny
EarlHall Tiny
Lewisohn Tiny
Philosophy Tiny
Buell Minuscule
AlmaMater Smallest
Dodge Small
Kent Small
CollegeWalk Gigantic
Journalism&Furnald Medium
Hamilton&Hartley&Wallach&JohnJay Largest
Lerner Medium
ButlerLibrary Large
Carman Small
```

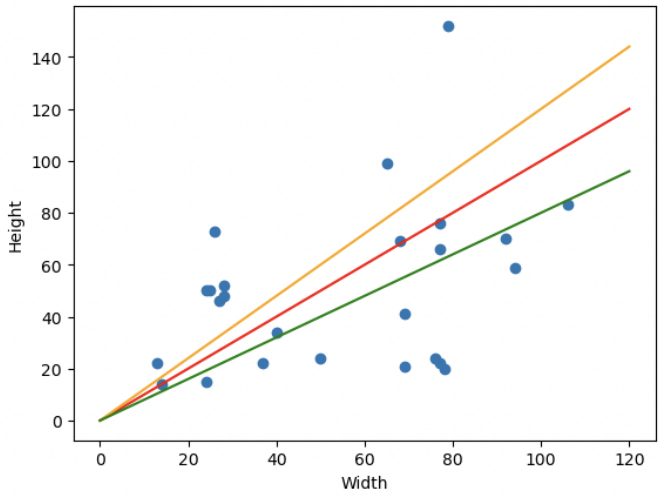
```
# the following few code blocks are used to determine the magic numbers used for the aspect ratio shape functions

# in this block I display area, diagonal length, name tuples sorted from largest area to least
aspect_list = list()
for key, val in raw_data.items():
    name, layer, com, area, ul, lr, diag, overlap = val

    width = lr[0] - ul[0]
    height = lr[1] - ul[1]

    aspect_list.append([round(width/height,1), width, height, name])

aspect_list.sort()

print('W/H', 'W ', 'H ', 'Name')
aspect_points = list()
for x in aspect_list:
    print(x[0], x[1], x[2], x[3])
    aspect_points.append([x[1], x[2]])
np_aspect_points = np.array(aspect_points)
```

```
W/H W  H  Name
0.4 26 73 NorthwestCorner
0.5 24 50 Avery
0.5 25 50 Fayerweather
0.5 28 52 Lewisohn
0.5 79 152 Hamilton&Hartley&Wallach&JohnJay
0.6 13 22 OldComputerCenter
0.6 27 46 Philosophy
0.6 28 48 Mathematics
0.7 65 99 Uris
1.0 14 14 AlmaMater
1.0 68 69 LowLibrary
1.0 77 76 Journalism&Furnald
1.2 40 34 SchapiroCEPSR
1.2 77 66 Chandler&Havemeyer
1.3 92 70 Schermerhorn
1.3 106 83 Mudd&EngTerrace&Fairchild&CS
1.6 24 15 Buell
1.6 94 59 ButlerLibrary
1.7 37 22 EarlHall
1.7 69 41 Lerner
2.1 50 24 StPaulChapel
3.2 76 24 Pupin
3.3 69 21 Carman
3.5 77 22 Dodge
3.9 78 20 Kent
16.1 274 17 CollegeWalk
```

```
# plotting relationship between building width and height to see if any clusters arrise

plt.scatter(np_aspect_points[:,0], np_aspect_points[:,1])
x = np.linspace(0,150)
y = x
plt.plot(x,y, color = 'r')
plt.xlabel("Width")
plt.ylabel("Height")
plt.show()
```




```
# constructing a plot without the outlier of college walk to see more clearly if any clusters arrise:
#plt.scatter(np_aspect_points[:,0], np_aspect_points[:,1])
plt.scatter(np_aspect_points[:len(np_aspect_points)-1,0], np_aspect_points[:len(np_aspect_points)-1,1])
x = np.linspace(0,120)
y = x
plt.plot(x,y, color = 'r')
y2 = 1.2*x  #x + 15
y3 = 0.8*x   #x - 20
plt.plot(x,y2, color = 'orange')
plt.plot(x,y3, color = 'g')
plt.xlabel("Width")
plt.ylabel("Height")
plt.show()
```



```
# ASPECT RATIO

# the following two functions use the width:height ratio of the buildings to determine whether
# the inputted building is the narrowest or widest or neither

def isNarrowest(raw_data, building_num):
    ul = raw_data[building_num][4]
    lr = raw_data[building_num][5]
    width = lr[0] - ul[0]
    height = lr[1] - ul[1]
    ratio = round(width/height,1)

    for key, val in raw_data.items():
        ul2 = val[4]
        lr2 = val[5]
        width2 = lr2[0] - ul2[0]
        height2 = lr2[1] - ul2[1]
        ratio2 = round(width2/height2,1)
        if ratio2 < ratio:
            return False

    return True

def isWidest(raw_data, building_num):
    ul = raw_data[building_num][4]
    lr = raw_data[building_num][5]
    width = lr[0] - ul[0]
    height = lr[1] - ul[1]
    ratio = round(width/height,1)

    for key, val in raw_data.items():
        ul2 = val[4]
        lr2 = val[5]
        width2 = lr2[0] - ul2[0]
        height2 = lr2[1] - ul2[1]
        ratio2 = round(width2/height2,1)
        if ratio2 > ratio:
            return False

    return True

# the following function uses the magic numbers determined from the above graphs to label
# a building as either Narrow, Wide, or Medium-Width
def aspectRatio(raw_data, building_num):
    # input: raw_data and building number
    # output: string labeling the aspectRatio of the inputted building

    ul = raw_data[building_num][4]
    lr = raw_data[building_num][5]
    width = lr[0] - ul[0]
    height = lr[1] - ul[1]
    ratio = round(width/height,1)
    #print(ratio)

    if isNarrowest(raw_data, building_num):
        return 'Narrowest'

    if isWidest(raw_data, building_num):
        return 'Widest'

    if ratio < 0.8:
        return 'Narrow'

    if ratio > 1.2:
        return 'Wide'

    return 'Medium-Width'

# testing the above aspect ratio functions:
for key, val in raw_data.items():
    print(val[0], aspectRatio(raw_data, key))
```

```
Pupin Wide
SchapiroCEPSR Medium-Width
Mudd&EngTerrace&Fairchild&CS Wide
NorthwestCorner Narrowest
Uris Narrow
Schermerhorn Wide
Chandler&Havemeyer Medium-Width
OldComputerCenter Narrow
Avery Narrow
Fayerweather Narrow
Mathematics Narrow
LowLibrary Medium-Width
StPaulChapel Wide
EarlHall Wide
Lewisohn Narrow
Philosophy Narrow
Buell Wide
AlmaMater Medium-Width
Dodge Wide
Kent Wide
CollegeWalk Widest
Journalism&Furnald Medium-Width
Hamilton&Hartley&Wallach&JohnJay Narrow
Lerner Wide
ButlerLibrary Wide
Carman Wide
```

```python
# GEOMETRY

def x_sym_score(mbr):
    # input: - mbr: 2D array depicting the splice of the total
    #               image that is contained within the mbr of the input building
    #        - area of building
    # return: percentage of symmetry

    tot = 0
    area = len(mbr) * len(mbr[0])

    for row in range(len(mbr)):
        for col in range(len(mbr[0])//2):
            if mbr[row,col] == mbr[row,len(mbr[0]) - col - 1]:
                tot += 1

    if len(mbr[0]) % 2 == 1:
        area = (area - len(mbr)) / 2 # area minus middle col
    else:
        area = area / 2
    return tot / area

def y_sym_score(mbr):
    # same as x_sym_score except it checks for vertical symetry not horizontal

    area = len(mbr) * len(mbr[0])
    tot = 0
    for col in range(len(mbr[0])):
        for row in range(len(mbr)//2):
            if mbr[row,col] == mbr[len(mbr)-row-1,col]:
                tot += 1

    if len(mbr) % 2 == 1:
        area = (area - len(mbr[0])) / 2 # area minus middle row
    else:
        area = area / 2

    return tot / area

# the outputted table will help give a sense of what a good symmetry score is as I will compare the returned
# percentages with the actual building shapes
# -> particularly, buildings like Low, Butler, and Uris should reflect high horizontal symmtery scores
#    while Havermeyer for example should reflect a relatively low score

# n = 9 #255 #113 #9 #246

print("{:<32} {:<6} {:<6}"
          .format('NAME', 'X SYM', 'Y SYM'))

for key, val in raw_data.items():
    ul = val[4]
    lr = val[5]
    layer = val[1]
    name = val[0]

    mbr = layer[ul[1]:lr[1], ul[0]:lr[0]]

    print("{:<32} {:<6} {:<6}" .format(name, round(x_sym_score(mbr),2),
                                       round(y_sym_score(mbr),2)))
```

NAME	X SYM	Y SYM
Pupin	0.92	0.78
SchapiroCEPSR	1.0	1.0
Mudd&EngTerrace&Fairchild&CS	0.58	0.47
NorthwestCorner	1.0	1.0
Uris	0.98	0.91
Schermerhorn	0.7	0.42
Chandler&Havemeyer	0.71	0.55
OldComputerCenter	1.0	1.0
Avery	0.82	1.0
Fayerweather	0.95	0.99
Mathematics	0.97	0.99
LowLibrary	0.99	0.99
StPaulChapel	0.86	0.96
EarlHall	0.97	0.99
Lewisohn	0.97	0.99
Philosophy	1.0	0.98
Buell	0.97	0.95
AlmaMater	1.0	1.0
Dodge	0.99	0.76
Kent	0.99	0.76
CollegeWalk	1.0	1.0
Journalism&Furnald	0.62	0.62
Hamilton&Hartley&Wallach&JohnJay	0.66	0.82
Lerner	1.0	1.0
ButlerLibrary	0.99	1.0
Carman	1.0	1.0

```
# buildings to me that seem vertically symmetrical (fold top over bottom)
# Noco, shapiro, old computer center, Fayerweather, mathematics, low, st paul chapel, earl,
# lewison, phiosophy, alma mater, college walk, buell, lerner, butler, carman, avery
# the minimum y-sym symmtery score of all of these is: 0.95

# buildings to me that seem horizontally symmetrical (fold left over right)
# noco, shapiro, computer center, buell, low, alma mater, earl, dodge, kent, college walk,
# lerner, butler, carmen
# kind of: uris, fayerweather, math, philosophy
# minimum x-sym symmetry score of all of these is: 0.95 (including ones that are not
# perfectly symetrical but are considered by me to be close enough)


# the magic numbers of 0.95 in the following two functions were determined from the
# observations from the previous two code cells

def isRectangle(br):
    # returns boolean stating whether or not the building is considered rectangular
    # (meaning it is symmetrical in x and in y)

#     val = raw_data[building_num]
#     ul = val[4]
#     lr = val[5]
#     layer = val[1]
#     mbr = layer[ul[1]:lr[1], ul[0]:lr[0]]

    return (x_sym_score(br) >= 0.95 and y_sym_score(br) >= 0.95)

def isSquare(br):
    # returns boolean stating whether or not the building is considered square-shaped
    # (meaning it is symmtrical in x and in y)

#     val = raw_data[building_num]
#     ul = val[4]
#     lr = val[5]
#    height = lr[1] - ul[1]
#     width = lr[0] - ul[0]

    height = br.shape[0]
    width = br.shape[1]

    # the only buildings I count as being square are low library, alma mater,
    # and Shapiro: their height / width values are the following, respectively:
    # 1.015, 1.0, and 0.85
    # because of these values I will define square as having a height:width ratio
    # that is within 0.15 of 1, inclusive

    ratio = height/width
    return isRectangle(mbr) and (ratio >= 0.85 and ratio <= 1.15)
```

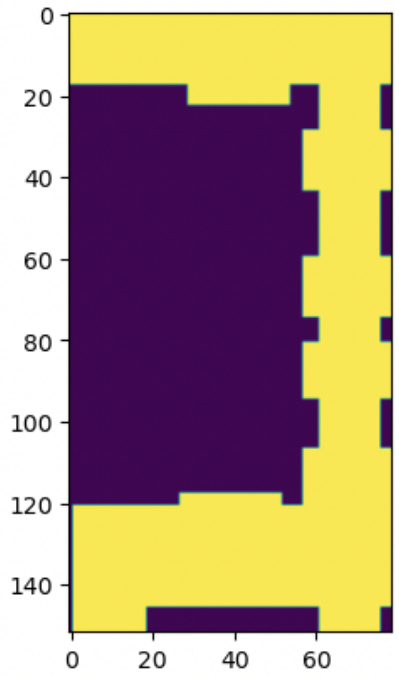
```
print("{:<32} {:<6} {:<6}"
          .format('NAME', 'SQUARE', 'RECT'))

for key, val in raw_data.items():
    val = raw_data[key]
    ul = val[4]
    lr = val[5]
    layer = val[1]
    mbr = layer[ul[1]:lr[1], ul[0]:lr[0]]

    squ = isSquare(mbr)
    rect = isRectangle(mbr)
    name = val[0]

    print("{:<32} {:<6} {:<6}" .format(name, str(squ), str(rect)))
```

```
NAME                             SQUARE RECT
Pupin                            False  False
SchapiroCEPSR                    True   True
Mudd&EngTerrace&Fairchild&CS     False  False
NorthwestCorner                  False  True
Uris                             False  False
Schermerhorn                     False  False
Chandler&Havemeyer               False  False
OldComputerCenter                False  True
Avery                            False  False
Fayerweather                     False  True
Mathematics                      False  True
LowLibrary                       True   True
StPaulChapel                     False  False
EarlHall                         False  True
Lewisohn                         False  True
Philosophy                       False  True
Buell                            False  False
AlmaMater                        True   True
Dodge                            False  False
Kent                             False  False
CollegeWalk                      False  True
Journalism&Furnald               False  False
Hamilton&Hartley&Wallach&JohnJay False  False
Lerner                           False  True
ButlerLibrary                    False  True
Carman                           False  True
```

```
def foldVert(binary_shape): # folds bottom of inputted 2d np binary array over the top over the center row
    width = binary_shape.shape[1]
    height = binary_shape.shape[0]
    res = np.zeros((height//2, width))
    for row in range(height//2):
        for col in range(width):
            if binary_shape[row,col] == 1 or binary_shape[height-row-1, col] == 1:
                res[row, col] = 1

    return res

def foldHoriz(binary_shape): # folds bottom of inputted 2d np binary array over the top over the center row
    width = binary_shape.shape[1]
    height = binary_shape.shape[0]
    res = np.zeros((height, width//2))
    for row in range(height):
        for col in range(width//2):
            if binary_shape[row,col] == 1 or binary_shape[row, width-col-1] == 1:
                res[row, col] = 1

    return res

val = raw_data[217]
ul = val[4]
lr = val[5]
layer = val[1]
mbr = layer[ul[1]:lr[1], ul[0]:lr[0]]
```

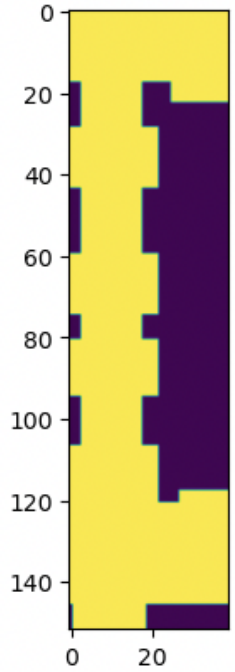
```
mbr = layer[ul[1]:lr[1], ul[0]:lr[0]]
plt.imshow(mbr)
```

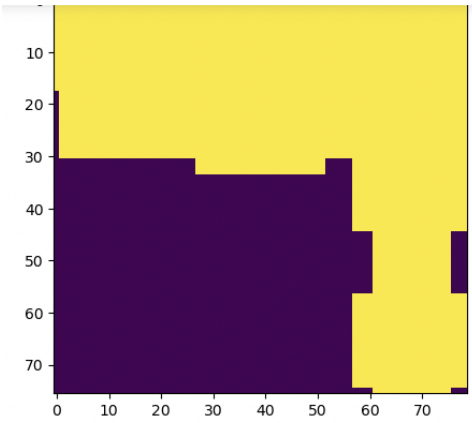
```
<matplotlib.image.AxesImage at 0x7fe17805f550>
```




```
plt.imshow(foldHoriz(mbr))
```

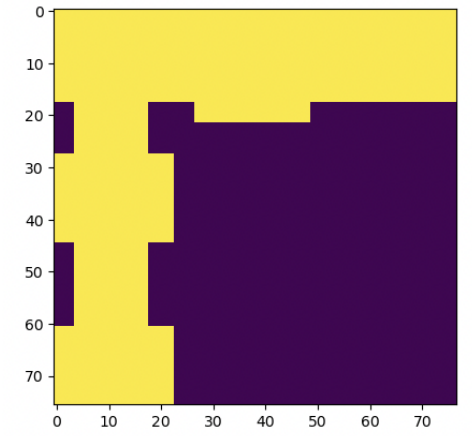
```
<matplotlib.image.AxesImage at 0x7fe1ca23c4c0>
```





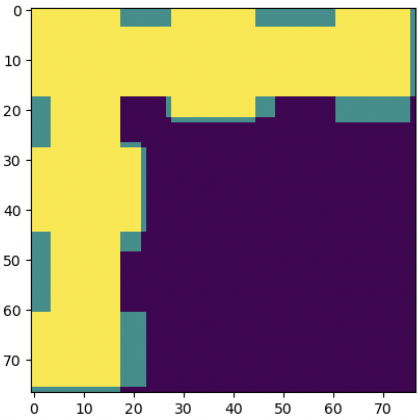

```
# the following three code blocks were used to determine how to fold building
# MBR's over both diagonals
# -> these folding operations are used to determine the overlap that results from
    # the folding -> ulitmately high overlap over one corner and low overlap in the other
    # while the input building is definately not a square or rectangle means that
    # the building must be L-shaped

val = raw_data[208] # L-shaped: 66, 208 #uris 47 #179 dodge
ul = val[4]
lr = val[5]
layer = val[1]
mbr = layer[ul[1]:lr[1], ul[0]:lr[0]]
#mbr = foldVert(mbr)
#mbr = foldHoriz(mbr)

plt.imshow(mbr)
```

```
<matplotlib.image.AxesImage at 0x7fe1cdb8c460>
```



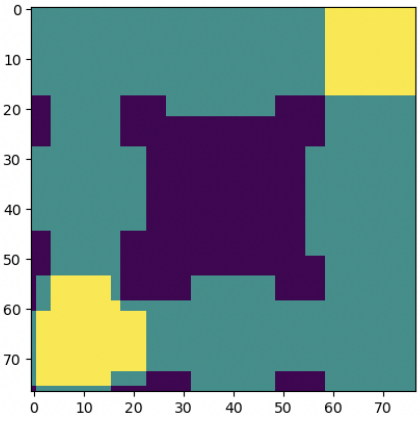

```
t = mbr.T

new_shape = np.zeros((max(t.shape[0], mbr.shape[0]), max(t.shape[1], mbr.shape[1])))
#new_shape = np.zeros(max(t.shape, mbr.shape))
for row in range(new_shape.shape[0]):
    for col in range(new_shape.shape[1]):
        if row < t.shape[0] and col < t.shape[1]:
            if t[row,col] == 1:
                new_shape[row, col] += 1
        if row < mbr.shape[0] and col < mbr.shape[1]:
            if mbr[row, col] == 1:
                new_shape[row, col] += 1

plt.imshow(new_shape)
print(overlap_qual(new_shape))
```

```
0.83
```




```
t2 = np.flip(mbr).T
new_shape2 = np.zeros((max(t2.shape[0], mbr.shape[0]), max(t2.shape[1], mbr.shape[1])))
x_shift = max(t2.shape[1], mbr.shape[1]) - min(t2.shape[1], mbr.shape[1])
#y_shift = max(t2.shape[0], mbr.shape[0]) - min(t2.shape[0], mbr.shape[0])
#print(y_shift+t2.shape[])

for row in range(new_shape2.shape[0]):
    for col in range(new_shape2.shape[1]):

        if row < t2.shape[0] and col < t2.shape[1]:
            if t2[row,col] == 1:# and col+x_shift < new_shape2.shape[1]:
                if mbr.shape[1] > mbr.shape[0]:
                    new_shape2[row, col+x_shift] += 1
                else:
                    new_shape2[row, col] += 1

        if row < mbr.shape[0] and col < mbr.shape[1]:
            if mbr[row, col] == 1:
                if mbr.shape[0] > mbr.shape[1]:
                    new_shape2[row, col+x_shift] += 1
                else:
                    new_shape2[row, col] += 1

plt.imshow(new_shape2)
print(overlap_qual(new_shape2))
```

```
0.16
```



```
def flipDiag1(br):
    t = br.T
    new_shape = np.zeros((max(t.shape[0], br.shape[0]), max(t.shape[1], br.shape[1])))
    for row in range(new_shape.shape[0]):
        for col in range(new_shape.shape[1]):
            if row < t.shape[0] and col < t.shape[1]:
                if t[row,col] == 1:
                    new_shape[row, col] += 1
            if row < br.shape[0] and col < br.shape[1]:
                if br[row, col] == 1:
                    new_shape[row, col] += 1

    return new_shape

def flipDiag2(br):
    t2 = np.flip(br).T
    new_shape2 = np.zeros((max(t2.shape[0], br.shape[0]), max(t2.shape[1], br.shape[1])))
    x_shift = max(t2.shape[1], br.shape[1]) - min(t2.shape[1], br.shape[1])

    for row in range(new_shape2.shape[0]):
        for col in range(new_shape2.shape[1]):
            if row < t2.shape[0] and col < t2.shape[1]:
                if t2[row,col] == 1:
                    if br.shape[1] > br.shape[0]:
                        new_shape2[row, col+x_shift] += 1
                    else:
                        new_shape2[row, col] += 1

            if row < br.shape[0] and col < br.shape[1]:
                if br[row, col] == 1:
                    if br.shape[0] > br.shape[1]:
                        new_shape2[row, col+x_shift] += 1
                    else:
                        new_shape2[row, col] += 1

    return new_shape2

def overlap_qual(br):
    tot_area = 0
    overlap_area = 0
    for row in range(br.shape[0]):
        for col in range(br.shape[1]):
            if br[row, col] == 2:
                overlap_area += 1
            if br[row, col] > 0:
                tot_area += 1

    return round(float(overlap_area) / (tot_area),2)
```

```
# this printed table will be used to determine the magic number of what percent overlap
# will determine the cut off for an L-shaped building

print("{:<37} {:<20} {:<20}"
          .format('NAME', 'Diag1 Overlap', 'Diag2 Overlap'))

for key, val in raw_data.items():
    val = raw_data[key]
    ul = val[4]
    lr = val[5]
    layer = val[1]
    mbr = layer[ul[1]:lr[1], ul[0]:lr[0]]

    oq1 = overlap_qual(flipDiag1(mbr))
    oq2 = overlap_qual(flipDiag2(mbr))

    if isSquare(mbr) or isRectangle(mbr):
        oq1 = 0
        oq2 = 0

    print("{:<37} {:<20} {:<20}" .format(val[0], oq1, oq2))

    #if val[0] in ['Lewisohn', 'Mathematics', 'Philosophy']:
    mbr2 = foldVert(foldHoriz(mbr))
    oq1 = overlap_qual(flipDiag1(mbr2))
    oq2 = overlap_qual(flipDiag2(mbr2))
    print("{:<37} {:<20} {:<20}" .format(val[0] + ' (2 fold)', oq1, oq2))

   # if val[0] == 'Hamilton&Hartley&Wallach&JohnJay':
    mbr3 = foldVert(mbr)
    oq1 = overlap_qual(flipDiag1(mbr3))
    oq2 = overlap_qual(flipDiag2(mbr3))
    print("{:<37} {:<20} {:<20}" .format(val[0] + ' (1 fold)', oq1, oq2))
```

NAME	Diag1 Overlap	Diag2 Overlap
Pupin	0.17	0.14
Pupin (2 fold)	0.19	0.19
Pupin (1 fold)	0.09	0.07
SchapiroCEPSR	0	0
SchapiroCEPSR (2 fold)	0.74	0.74
SchapiroCEPSR (1 fold)	0.27	0.27
Mudd&EngTerrace&Fairchild&CS	0.26	0.47
Mudd&EngTerrace&Fairchild&CS (2 fold)	0.63	0.63
Mudd&EngTerrace&Fairchild&CS (1 fold)	0.17	0.27
NorthwestCorner	0	0
NorthwestCorner (2 fold)	0.22	0.22
NorthwestCorner (1 fold)	0.57	0.57
Uris	0.43	0.43
Uris (2 fold)	0.43	0.37
Uris (1 fold)	0.62	0.62
Schermerhorn	0.18	0.28
Schermerhorn (2 fold)	0.62	0.56
Schermerhorn (1 fold)	0.19	0.23
Chandler&Havemeyer	0.38	0.36
Chandler&Havemeyer (2 fold)	0.75	0.72
Chandler&Havemeyer (1 fold)	0.26	0.25
OldComputerCenter	0	0
OldComputerCenter (2 fold)	0.38	0.38
OldComputerCenter (1 fold)	0.73	0.73
Avery	0.28	0.27
Avery (2 fold)	0.32	0.32
Avery (1 fold)	0.78	0.75
Fayerweather	0	0
Fayerweather (2 fold)	0.25	0.22
Fayerweather (1 fold)	0.9	0.85
Mathematics	0	0
Mathematics (2 fold)	0.47	0.48
Mathematics (1 fold)	0.69	0.69
LowLibrary	0	0
LowLibrary (2 fold)	0.98	0.65
LowLibrary (1 fold)	0.33	0.33
StPaulChapel	0.22	0.32
StPaulChapel (2 fold)	0.28	0.34
StPaulChapel (1 fold)	0.08	0.13
EarlHall	0	0
EarlHall (2 fold)	0.35	0.51
EarlHall (1 fold)	0.13	0.15
Lewisohn	0	0
Lewisohn (2 fold)	0.44	0.44
Lewisohn (1 fold)	0.72	0.73
Philosophy	0	0
Philosophy (2 fold)	0.51	0.51
Philosophy (1 fold)	0.65	0.65
Buell	0.41	0.44
Buell (2 fold)	0.32	0.48
Buell (1 fold)	0.12	0.14
AlmaMater	0	0
AlmaMater (2 fold)	1.0	1.0
AlmaMater (1 fold)	0.33	0.33
Dodge	0.19	0.19
Dodge (2 fold)	0.17	0.17
Dodge (1 fold)	0.08	0.08
Kent	0.17	0.17
Kent (2 fold)	0.15	0.15
Kent (1 fold)	0.07	0.07
CollegeWalk	0	0
CollegeWalk (2 fold)	0.03	0.03
CollegeWalk (1 fold)	0.01	0.01
Journalism&Furnald	0.83	0.16
Journalism&Furnald (2 fold)	0.91	0.53
Journalism&Furnald (1 fold)	0.39	0.1
Hamilton&Hartley&Wallach&JohnJay	0.07	0.27
Hamilton&Hartley&Wallach&JohnJay (2 fold)	0.46	0.45
Hamilton&Hartley&Wallach&JohnJay (1 fold)	0.24	0.69
Lerner	0	0
Lerner (2 fold)	0.42	0.42
Lerner (1 fold)	0.17	0.17
ButlerLibrary	0	0
ButlerLibrary (2 fold)	0.41	0.49
ButlerLibrary (1 fold)	0.17	0.17
Carman	0	0
Carman (2 fold)	0.17	0.17
Carman (1 fold)	0.08	0.08

```
def isLShaped2(br):
    if isSquare(br) or isRectangle(br):
        return False

    d1 = flipDiag1(br)
    d2 = flipDiag2(br)

    # for the 2-fold I-shaped buildings:
    if overlap_qual(d1) >= 0.44 and overlap_qual(d1) >= 0.44:
        return True

    L_overlap = 0.69
    if (overlap_qual(d1) >= L_overlap and
        overlap_qual(d2) < L_overlap) or (overlap_qual(d2) >= L_overlap and overlap_
        return True

    return False
```

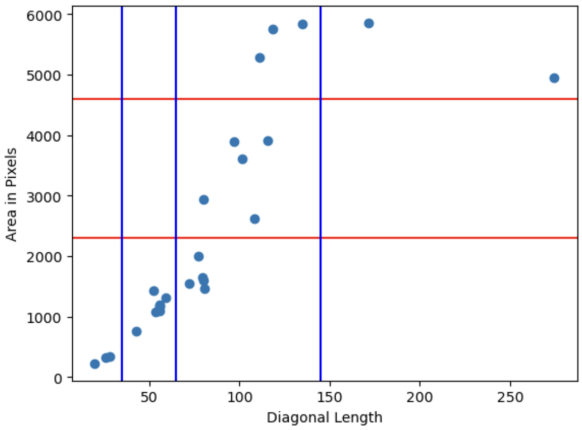
```
def isCShaped(br):
    horiz = foldHoriz(br)
    vert = foldVert(br)
    if isLShaped2(horiz) or isLShaped2(vert):
        return True
    return False

def isIShaped(br):
    horiz = foldHoriz(br)
    vert = foldVert(br)
    if isCShaped(horiz) or isCShaped(vert):
        return True
    return False

def geometry(raw_data, building_num):

    val = raw_data[building_num]
    ul = val[4]
    lr = val[5]
    layer = val[1]
    mbr_ = layer[ul[1]:lr[1], ul[0]:lr[0]]
    #plt.imshow(mbr_)

    if isSquare(mbr_):
        return "Square-shaped"
    if isRectangle(mbr_):

        if isIShaped(mbr_):
            return "I-shaped"

        return "Rectangle-shaped"

    if isLShaped(mbr_):
        return "L-shaped"
    if isCShaped(mbr_) and ():
        return "C-shaped"

    return "Asymmetrical"
```

```
# creating table using the above shape labeling functions

print("{:<32} {:<12} {:<14} {:<10}"
          .format('NAME', 'SIZE', 'ASPECT RATIO', 'GEOMETRY'))

for key, val in raw_data.items():
    print("{:<32} {:<12} {:<14} {:<10}" .format(str(val[0]), size_(raw_data, key),
                  aspectRatio(raw_data, key), geometry(raw_data, key)))
```

Discussion:

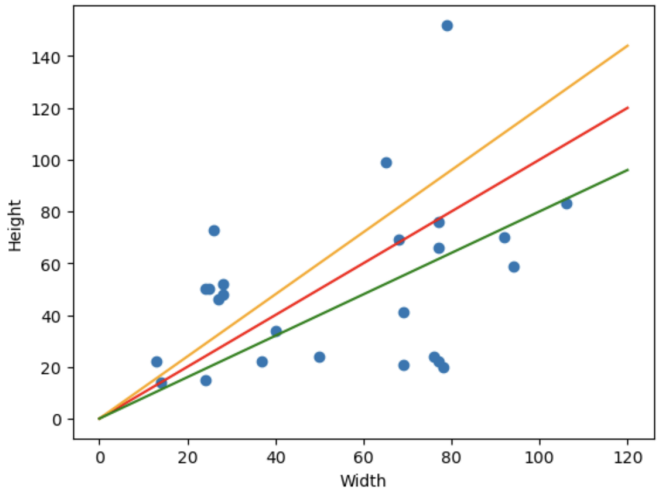
The first thing I did for this step was after determining that area and diagonal length were two attributes of the buildings that were good descriptors of size, I graphed a scatter plot of all of the building's (diagonal length, area) pairs. I did this to see if I could identify any clusters within the data that had to do with the varying sizes of the buildings. After plotting the scatter points I

also added some horizontal and vertical lines to the graph in locations that seemed to section off the point clusters based on how close the points seemed to be in my opinion. The graph of this is depicted below.



I decided that, since there appeared to be six distinct point clusters, to denote size using six different labels, instead of the three labels suggested by the instructions. These labels would be, from bottom left to top right, Minuscule, Tiny, Small, Medium, Large, and Gigantic. All the magic numbers used in the size_() function were determined from this graph and from my own personal judgment on the bounds of each of the (diagonal length, area) point clusters. I also created isSmallest() and isLargest() functions that compare the input buildings to all of the other buildings and returns a boolean if the input building is the smallest or largest of all of them, respectively. In testing these functions I observed that larger buildings are correctly labeled with a label denoting a larger size like ‘Large’ or ‘Gigantic’, and vice versa for smaller buildings. My isSmallest() and isLargest() functions correctly classify the smallest and largest buildings as ‘AlmaMater’ and ‘Hamilton&Hartley&Wallach&JohnJay’.

The next thing I did for this step was plot the relationship between building width and height (calculated from the upper left and lower right coordinates of the MBR) in order to find magic numbers I could use to differentiate the narrow, wide, and medium-width buildings from one another. After plotting the scatter plot of (width, height) and plotting a line representing width/height = 1, I messed around with the slope of two other lines to determine where a reasonable cut off of the width:height ratio would be such that the buildings with similar widths and heights were clearly differentiated from the buildings with very different widths and heights. This resulting plot is depicted below, where the slope of the orange line is 1.2 and the slope of the green line is 0.8.



It is important to note I chose to not include the outlier of College Walk in this scatter plot because including it increased the range of the x-axis of the graph, making the other points appear closer together, which made it more difficult to determine the values of slope I should use to decorate different narrownesses and widths of building (width, height) points. I used the magic numbers of 1.2 and 0.8 in my aspectRatio() function which returns a string labeling the inputted building as either 'Narrow', 'Wide', or 'Medium-Width'. I also created isNarrowest() and isWidest() functions, similar to the isSmallest() and isLargest() size functions, that take in a building and return a boolean value denoting whether the inputted building is the narrowest of all of the buildings or the widest of all or neither. In my observations of the aspectRatio() function I noticed that buildings with similar widths and heights were correctly labeled as 'Medium-Width', while buildings that were more tall than wide were labeled 'Narrow' and vice versa for buildings that are more wide than tall. The isNarrowest() and isWidest() functions also correctly classify the narrowest and widest buildings on campus as 'NorthwestCorner' and 'CollegeWalk'.

The next thing I did for this step was begin to classify the geometry of the buildings. I created functions that measured the symmetry of the building's MBR over both the vertical and horizontal axis. Then I used these two functions in my isSquare and isRectangle functions to determine if the input building was classified as one of these shapes. If the building had very similar MBR width and heights and was symmetrical in the x and y directions then the building is classified as square. If, alternatively, the building has differing width and height but was still symmetric in the x and y directions, the building is classified as rectangular. One thing to note is that I-shaped buildings, given this definition of rectangular should be identified as rectangular as well. The way I ensured that I-shaped buildings were identified as I-shaped and not rectangular was I check first if a building is I-shaped, and if so I classify it as such, before I check for rectangular-ness. The magic numbers of 0.95 in both my isSquare and isRectangle functions arise from the fact that if the percentage of symmetry was higher than 95%, I want to label this building as symmetrical. 0.95 itself was determined by comparing my ground truth list of square

and rectangular buildings with their x and y symmetry scores, as I observed that square and rectangular buildings all tended to have x and y symmetry scores greater than or equal to 0.95.

Next, I created isIShaped which folds the input bounding rectangle horizontally and vertically and inserts each of these folded shapes into isCShaped separately. If one of these folded bounding rectangles are identified to be C-shaped, then isIShaped returns True, returning False otherwise. I then wrote isCShaped, which follows a similar algorithm to isIShaped. If either the vertical or horizontal fold of the input bounding rectangle of isCShaped returns True when inputted into isLShaped, then isCShaped returns True, returning False otherwise. Finally, I then wrote isLShaped. To do so I created two functions that fold the inputted bounding rectangle over its two diagonals separately- I used the np transpose function to get both folds. I then created a function that measured the percentage of overlap from these diagonal folds using total pixel area from both objects and the overlap pixel area. To determine what I should use for the cutoff point for determining if an object was indeed L-shaped I printed out the overlap quality of each building over each diagonal as well as the overlap over each diagonal of each building folded over both one and two axes. After observing this I determined my magic numbers would be 0.44 for the overlap qualities of both diagonals for I-shaped buildings specifically, and 0.69 for the overlap qualities of one or another diagonal (but not both, since L-shaped buildings should only be symmetrical over one diagonal to be truly L-shaped). The following table shows all the shape qualities including geometry. With observation, one can see that the majority of the buildings are geometrically labeled accurately. A few, however, are inaccurately labeled, those being Mudd, Uris, Havermeyer, Buell, Journalism, and Hamilton. These errors are likely present because of the cutoff numbers I chose. To reduce the error in this algorithm I could continue testing overlap values for each and every building but I decided to move on with the rest of the assignment as a 77% success rate was sufficient for me for the geometric aspect of this step.

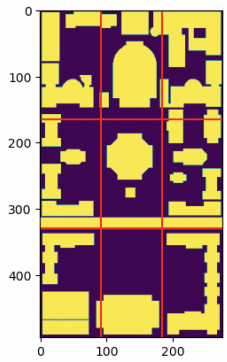
NAME	SIZE	ASPECT RATIO	GEOMETRY
Pupin	Small	Wide	Asymmetrical
SchapiroCEPSR	Tiny	Medium-Width	Square-shaped
Mudd&EngTerrace&Fairchild&CS	Large	Wide	C-shaped
NorthwestCorner	Small	Narrowest	Rectangle-shaped
Uris	Large	Narrow	C-shaped
Schermerhorn	Medium	Wide	Asymmetrical
Chandler&Havemeyer	Medium	Medium-Width	Square-shaped
OldComputerCenter	Minuscule	Narrow	Rectangle-shaped
Avery	Tiny	Narrow	Asymmetrical
Fayerweather	Tiny	Narrow	Rectangle-shaped
Mathematics	Tiny	Narrow	I-shaped
LowLibrary	Medium	Medium-Width	Square-shaped
StPaulChapel	Tiny	Wide	Asymmetrical
EarlHall	Tiny	Wide	Rectangle-shaped
Lewisohn	Tiny	Narrow	I-shaped
Philosophy	Tiny	Narrow	I-shaped
Buell	Minuscule	Wide	C-shaped
AlmaMater	Smallest	Medium-Width	Square-shaped
Dodge	Small	Wide	C-shaped
Kent	Small	Wide	Asymmetrical
CollegeWalk	Gigantic	Widest	Rectangle-shaped
Journalism&Furnald	Medium	Medium-Width	Square-shaped
Hamilton&Hartley&Wallach&JohnJay	Largest	Narrow	Asymmetrical
Lerner	Medium	Wide	Rectangle-shaped
ButlerLibrary	Large	Wide	Rectangle-shaped
Carman	Small	Wide	Rectangle-shaped

Step 3:

Code:

```
# Creating Numpy Array from 'Campus.pgm' file to determine the absolute location boundaries of buildings
campus_file = Image.open('Desktop/vis_int_hw3/Campus.pgm')
campus_arr = np.asarray(campus_file)
tot_height,tot_width = campus_arr.shape
plt.axhline(y = tot_height/3, color = 'r', linestyle = '-')
plt.axhline(y = 2*tot_height/3, color = 'r', linestyle = '-')
plt.axvline(x = tot_width/3, color = 'r', label = 'axvline - full height')
plt.axvline(x = 2*tot_width/3, color = 'r', label = 'axvline - full height')
plt.imshow(campus_arr)
```

```
<matplotlib.image.AxesImage at 0x7fe1e9b89e20>
```




```
# Next I will use the above grid and the COM of each of the buildings to construct functions that
# will labeled where buildings are vertically and horizontally in space as well as labeling the
# orientation of the building

# the following two functions determine whether the input building is the uppermost or lowermost building or neither
def isUppermost(raw_data, num):
    com_y = raw_data[num][2][1]
    for key, val in raw_data.items():
        if val[2][1] < com_y:
            return False
    return True

def isLowermost(raw_data, num):
    com_y = raw_data[num][2][1]
    for key, val in raw_data.items():
        if val[2][1] > com_y:
            return False
    return True

# the following function labels the vertical absolute location of the input building using the following
# labels: 'Uppermost', 'Upper', 'Mid-Height', 'Lower', 'Lowermost'
def vertical_location(raw_data, num, tot_height):
    com_y = raw_data[num][2][1]

    if isUppermost(raw_data, num):
        return 'Uppermost'
    if isLowermost(raw_data, num):
        return 'Lowermost'

    if com_y < tot_height/3:
        return 'Upper'

    if com_y < 2*tot_height/3:
        return 'Mid-Height'

    return 'Lower'

# the following two functions determine whether the input building is the leftmost or rightmost building or neither
def isLeftmost(raw_data, num):
    com_x = raw_data[num][2][0]
    for key, val in raw_data.items():
        if val[2][0] < com_x:
            return False
    return True
```

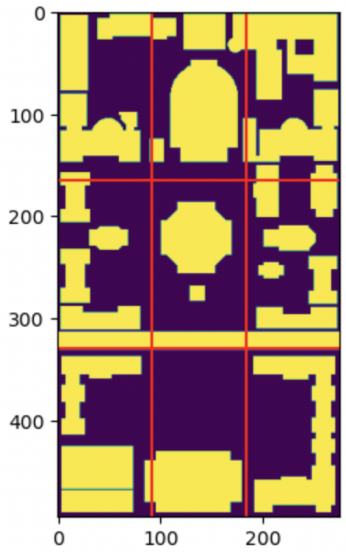
```
def isRightmost(raw_data, num):
    com_x = raw_data[num][2][0]
    for key, val in raw_data.items():
        if val[2][0] > com_x:
            return False
    return True

# the following function labels the horizontal absolute location of the input building using the following
# labels: 'Leftmost', 'Left', 'Mid-Width', 'Right', 'Rightmost'
def horizontal_location(raw_data, num, tot_width):
    com_x = raw_data[num][2][0]

    if isLeftmost(raw_data, num):
        return 'Leftmost'
    if isRightmost(raw_data, num):
        return 'Rightmost'

    if com_x < tot_width/3:
        return 'Left'

    if com_x < 2*tot_width/3:
        return 'Mid-Width'

    return 'Right'

# the following function labels the orientation of the input building based on which dimention, height or width,
# of the building is larger than the other
# (Note: the differentiation between this function and the aspect ratio function is that the aspect ratio function
# has more leeway in deciding whether a building is Medium-wide, while in the following orientation function,
# the only way 'Non-oriented' is returned as a label is if building width is exactly equal to building height)
def orientation(raw_data, num):
    ul = raw_data[num][4]
    lr = raw_data[num][5]
    w = lr[0] - ul[0]
    h = lr[1] - ul[1]
    if w == h:
        return 'Non-Oriented'

    if w < h:
        return 'Vertically-Oriented'

    return 'Horizontally-Oriented'

# creating tabling using the above absolute location functions
# also creating a dictionary that stores a tuple of (vertical location, horizontal location, and orientation)
# as its keys, with its values being a corresponding list of the buildings that identify with these attributes

abs_loc_confusion = dict()

print("{:<32} {:<12} {:<14} {:<10}"
          .format('NAME', 'VERTICALLY', 'HORIZONTALLY', 'ORIENTED'))

for key, val in raw_data.items():
    vloc = vertical_location(raw_data, key, tot_height)
    hloc = horizontal_location(raw_data, key, tot_width)
    ori = orientation(raw_data, key)
    print("{:<32} {:<12} {:<14} {:<10}" .format(str(val[0]), vloc, hloc, ori))

    abs_loc = (vloc, hloc, ori)
    if abs_loc in abs_loc_confusion.keys():
        abs_loc_confusion[abs_loc].append(val[0])
    else:
        abs_loc_confusion[abs_loc] = [val[0]]
```

```
NAME                                VERTICALLY   HORIZONTALLY   ORIENTED
Pupin                               Uppermost    Left           Horizontally-Oriented
SchapiroCEPSR                       Upper        Mid-Width      Horizontally-Oriented
Mudd&EngTerrace&Fairchild&CS        Upper        Right          Horizontally-Oriented
NorthwestCorner                     Upper        Leftmost       Vertically-Oriented
Uris                                Upper        Mid-Width      Vertically-Oriented
Schermerhorn                        Upper        Right          Horizontally-Oriented
Chandler&Havemeyer                  Upper        Left           Horizontally-Oriented
OldComputerCenter                   Upper        Mid-Width      Vertically-Oriented
Avery                               Mid-Height   Right          Vertically-Oriented
Fayerweather                        Mid-Height   Rightmost      Vertically-Oriented
Mathematics                         Mid-Height   Left           Vertically-Oriented
LowLibrary                          Mid-Height   Mid-Width      Vertically-Oriented
StPaulChapel                        Mid-Height   Right          Horizontally-Oriented
EarlHall                            Mid-Height   Left           Horizontally-Oriented
Lewisohn                            Mid-Height   Left           Vertically-Oriented
Philosophy                          Mid-Height   Right          Vertically-Oriented
Buell                               Mid-Height   Right          Horizontally-Oriented
AlmaMater                           Mid-Height   Mid-Width      Non-Oriented
Dodge                               Mid-Height   Left           Horizontally-Oriented
Kent                                Mid-Height   Right          Horizontally-Oriented
CollegeWalk                         Mid-Height   Mid-Width      Horizontally-Oriented
Journalism&Furnald                  Lower        Left           Horizontally-Oriented
Hamilton&Hartley&Wallach&JohnJay    Lower        Right          Vertically-Oriented
Lerner                              Lower        Left           Horizontally-Oriented
ButlerLibrary                       Lower        Mid-Width      Horizontally-Oriented
Carman                              Lowermost    Left           Horizontally-Oriented
```

```
# printing final table for absolute location step:

print("{:<32} {:<12} {:<14} {:<23} {:<10}"
          .format('NAME', 'VERTICALLY', 'HORIZONTALLY', 'ORIENTED', 'CONFUSION'))

# for key, val in raw_data.items():
#     vloc = vertical_location(raw_data, key, tot_height)
#     hloc = horizontal_location(raw_data, key, tot_width)
#     ori = orientation(raw_data, key)
#     print("{:<32} {:<12} {:<14} {:<10}" .format(str(val[0]), vloc, hloc, ori))

#     abs_loc = (vloc, hloc, ori)
#     if abs_loc in abs_loc_confusion.keys():
#         abs_loc_confusion[abs_loc].append(val[0])
#     else:
#         abs_loc_confusion[abs_loc] = [val[0]]

for key, val in raw_data.items():
    vloc = vertical_location(raw_data, key, tot_height)
    hloc = horizontal_location(raw_data, key, tot_width)
    ori = orientation(raw_data, key)
    print("{:<32} {:<12} {:<14} {:<23}" .format(str(val[0]), vloc, hloc, ori), end=" ")
    if len(abs_loc_confusion[(vloc, hloc, ori)]) > 1:
        for x in abs_loc_confusion[(vloc, hloc, ori)]:
            if x != val[0]:
                print(x, end=" ")
    print()
```

Discussion:

For this step I first divided the over campus image height and width into three sections each, as depicted in the following image, in order to classify the absolute locations around campus.



I then created isUppermost(), isLowermost(), isLeftmost(), and isRightmost() functions using the building center of mass coordinates. Next, I utilized the 9 resulting sections of the width and height division along with these four helper functions to determine the vertical and horizontal locations of each building. I chose to uniformly divide campus into these 9 sections and not in a different way because usually when dealing with location and absolute location, locations are sectioned off in a uniform manner. The following table shows my results for this step.

NAME	VERTICALLY	HORIZONTALLY	ORIENTED
Pupin	Uppermost	Left	Horizontally-Oriented
SchapiroCEPSR	Upper	Mid_Width	Horizontally-Oriented
Mudd&EngTerrace&Fairchild&CS	Upper	Right	Horizontally-Oriented
NorthwestCorner	Upper	Leftmost	Vertically-Oriented
Uris	Upper	Mid_Width	Vertically-Oriented
Schermerhorn	Upper	Right	Horizontally-Oriented
Chandler&Havemeyer	Upper	Left	Horizontally-Oriented
OldComputerCenter	Upper	Mid_Width	Vertically-Oriented
Avery	Mid-Height	Right	Vertically-Oriented
Fayerweather	Mid-Height	Rightmost	Vertically-Oriented
Mathematics	Mid-Height	Left	Vertically-Oriented
LowLibrary	Mid-Height	Mid_Width	Vertically-Oriented
StPaulChapel	Mid-Height	Right	Horizontally-Oriented
EarlHall	Mid-Height	Left	Horizontally-Oriented
Lewisohn	Mid-Height	Left	Vertically-Oriented
Philosophy	Mid-Height	Right	Vertically-Oriented
Buell	Mid-Height	Right	Horizontally-Oriented
AlmaMater	Mid-Height	Mid_Width	Non-Oriented
Dodge	Mid-Height	Left	Horizontally-Oriented
Kent	Mid-Height	Right	Horizontally-Oriented
CollegeWalk	Mid-Height	Mid_Width	Horizontally-Oriented
Journalism&Furnald	Lower	Left	Horizontally-Oriented
Hamilton&Hartley&Wallach&JohnJay	Lower	Right	Vertically-Oriented
Lerner	Lower	Left	Horizontally-Oriented
ButlerLibrary	Lower	Mid_Width	Horizontally-Oriented
Carman	Lowermost	Left	Horizontally-Oriented

The orientation function used image height and width to determine whether a building was horizontally or vertically oriented. If the height was larger than the width, it was labeled as ‘Vertically-Oriented’ and vice versa for ‘Horizontally-Oriented’, where the only way the ‘Non-Oriented’ label would be return was if the width and height of the building were exactly equivalent. From this table I observe correct action by the system. The buildings who’s center of

masses are in the upper left section of campus are labeled as such, and the same follows for the rest of the buildings and the 8 other absolute location sections. Horizontal and vertical orientation also appear to be correctly identified by the system. Overall, this step in its entirety resulted in a 100% success rate.

The following table includes a column for absolute location confusion:

NAME	VERTICALLY	HORIZONTALLY	ORIENTED	CONFUSION
Pupin	Uppermost	Left	Horizontally-Oriented	
SchapiroCEPSR	Upper	Mid_Width	Horizontally-Oriented	
Mudd&EngTerrace&Fairchild&CS	Upper	Right	Horizontally-Oriented	Schermerhorn
NorthwestCorner	Upper	Leftmost	Vertically-Oriented	
Uris	Upper	Mid_Width	Vertically-Oriented	OldComputerCenter
Schermerhorn	Upper	Right	Horizontally-Oriented	Mudd&EngTerrace&Fairchild&CS
Chandler&Havemeyer	Upper	Left	Horizontally-Oriented	
OldComputerCenter	Upper	Mid_Width	Vertically-Oriented	Uris
Avery	Mid-Height	Right	Vertically-Oriented	Philosophy
Fayerweather	Mid-Height	Rightmost	Vertically-Oriented	
Mathematics	Mid-Height	Left	Vertically-Oriented	Lewisohn
LowLibrary	Mid-Height	Mid_Width	Vertically-Oriented	
StPaulChapel	Mid-Height	Right	Horizontally-Oriented	Buell Kent
EarlHall	Mid-Height	Left	Horizontally-Oriented	Dodge
Lewisohn	Mid-Height	Left	Vertically-Oriented	Mathematics
Philosophy	Mid-Height	Right	Vertically-Oriented	Avery
Buell	Mid-Height	Right	Horizontally-Oriented	StPaulChapel Kent
AlmaMater	Mid-Height	Mid_Width	Non-Oriented	
Dodge	Mid-Height	Left	Horizontally-Oriented	EarlHall
Kent	Mid-Height	Right	Horizontally-Oriented	StPaulChapel Buell
CollegeWalk	Mid-Height	Mid_Width	Horizontally-Oriented	
Journalism&Furnald	Lower	Left	Horizontally-Oriented	Lerner
Hamilton&Hartley&Wallach&JohnJay	Lower	Right	Vertically-Oriented	
Lerner	Lower	Left	Horizontally-Oriented	Journalism&Furnald
ButlerLibrary	Lower	Mid_Width	Horizontally-Oriented	
Carman	Lowermost	Left	Horizontally-Oriented	

The Minimizations for each building (each shorter description, if any, reduces confusion specifically when comparing two buildings that share the same three descriptions) are shown in the table-like list below.

Pupin: No confusion
Shapiro: No confusion
Mudd&Eng…: Uppermost Right
NorthwestCorner: No confusion
Uris: Central-most vertically-oriented
Schmerhorn: Rightmost horizontally-oriented
Chandler&Hav: No confusion
OldComputerCenter: leftmost vertically-oriented
Avery: Uppermost vertically-oriented
Fayerweather: No confusion
Mathematics: Uppermost vertically-oriented
LowLibrary: No confusion
StPaulChapel: Uppermost horizontally-oriented
EarlHall: Uppermost horizontally-oriented
Lewisohn: Lowermost vertically-oriented
Philosophy: Lowermost vertically-oriented
Buell: leftmost horizontally-oriented
AlmaMater: No confusion

Dodge: Lowermost horizontally-oriented
Kent: Lowermost horizontally-oriented
CollegeWalk: No confusion
Journalism&Fur: Uppermost horizontally-oriented
Hamilton&Hartley&...: No confusion
Lerner: Lowermost horizontally-oriented
ButlerLibrary: No confusion
Carman: No confusion

Step 4:

Code:

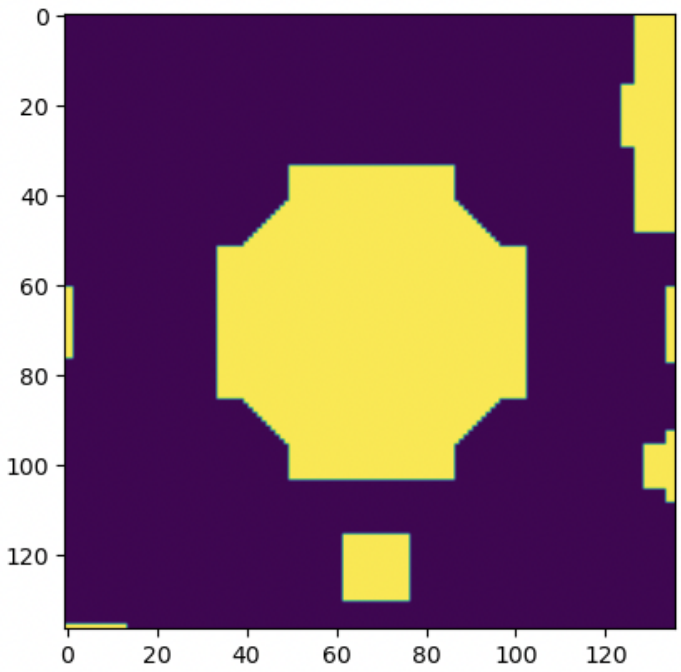
```python
# the following function determines the new mbr of a building whose dimensions are twice the size of the
# original mbr: this function is used to determine what buildings are nearby the inputted building as
# building's whose original mbr's intersect with the target building's new mbr are considered near the
# target image
def determine_nearby_mbr(raw_data, building_num):
    val = raw_data[building_num]
    x1 = val[4][0]
    x2 = val[5][0]
    y1 = val[4][1]
    y2 = val[5][1]
    w = x2 - x1
    h = y2 - y1
    #layer = val[1]

    # noting that the shape of the entire image is: (495, 275)

    if x1 - w/2 >= 0:
        x1 = x1 - w//2
    else:
        x1 = 0

    if x2 + w/2 < 275:
        x2 = x2 + w//2
    else:
        x2 = 274

    if y1 - h/2 >= 0:
        y1 = y1 - h//2
    else:
        y1 = 0

    if y2 + h/2 < 495:
        y2 = y2 + h//2
    else:
        y2 = 494

    #mbr = layer[y1:y2, x1:x2]
    return x1, x2, y1, y2

def intersecting_mbrs(mbr1_ul, mbr1_lr, mbr2_ul, mbr2_lr):

    intersects = True

    # If one rectangle is on left side of other
    if mbr1_ul[0] >= mbr2_lr[0] or mbr2_ul[0] >= mbr1_lr[0]:
#         print('1')
#         print(mbr1_ul[0], mbr2_lr[0])
#         print(mbr2_ul[0], mbr1_lr[0])
        intersects = False

    # If one rectangle is above other
    if mbr1_lr[1] <= mbr2_ul[1] or mbr2_lr[1] <= mbr1_ul[1]:
#         print('2')
#         print(mbr1_lr[1], mbr2_ul[1])
#         print(mbr2_lr[1], mbr1_ul[1])
        intersects = False

    return intersects

def nearTo(raw_data, S_num, T_num):
    x1, x2, y1, y2 = determine_nearby_mbr(raw_data, S_num)
    S_mbr_ul = (x1,y1)
    S_mbr_lr = (x2,y2)

    T = raw_data[T_num]
    T_mbr_ul = T[4]
    T_mbr_lr = T[5]

    return intersecting_mbrs(S_mbr_ul, S_mbr_lr, T_mbr_ul, T_mbr_lr)
```

```
n1 = 189
n2 = 217

print(raw_data[n1][0], raw_data[n2][0])

ul2 = raw_data[n2][4]
lr2 = raw_data[n2][5]

x1, x2, y1, y2 = determine_nearby_mbr(raw_data, n1)

print(ul2[0],lr2[0],ul2[1],lr2[1])
print(x1,x2,y1,y2)

print(intersecting_mbrs((x1,y1), (x2,y2), ul2, lr2))
print(intersecting_mbrs(ul2, lr2, (x1,y1), (x2,y2)))
```

```
Kent Hamilton&Hartley&Wallach&JohnJay
191 270 338 490
155 274 280 320
False
False
```

```
n = 113
x1, x2, y1, y2 = determine_nearby_mbr(raw_data, n)
#layer = raw_data[n][1]
#plt.imshow(layer[y1:y2, x1:x2])
print(x1,x2,y1,y2)
plt.imshow(campus_arr[y1:y2, x1:x2])
```

```
67 203 153 290

<matplotlib.image.AxesImage at 0x7fe1c9e90a00>
```



```
near_sources_to_targets = dict() # keys are source buildings, values are target buildings near to the
                                 # source building
near_targets_to_sources = dict() # keys are target buildings, values are source buildings near to
                                 # the target building
for building1 in raw_data.keys():
    for building2 in raw_data.keys():
        if building1 == building2:
            continue

        if nearTo(raw_data, building1, building2):
            if raw_data[building1][0] in near_sources_to_targets.keys():
                near_sources_to_targets[raw_data[building1][0]].append(raw_data[building2][0])
            else:
                near_sources_to_targets[raw_data[building1][0]] = [raw_data[building2][0]]

        if nearTo(raw_data, building2, building1):
            if raw_data[building1][0] in near_targets_to_sources.keys():
                near_targets_to_sources[raw_data[building1][0]].append(raw_data[building2][0])
            else:
                near_targets_to_sources[raw_data[building1][0]] = [raw_data[building2][0]]

for k, v in raw_data.items():
    print(v[0])
    print('S to T: ', end = "")
    if v[0] in near_sources_to_targets.keys():
        for x in near_sources_to_targets[v[0]]:
            print(x, end = " ")
    print()
    print('T to S: ', end = "")
    if v[0] in near_targets_to_sources.keys():
        for x in near_targets_to_sources[v[0]]:
            print(x, end = " ")
    print()
    print()
```

Note: output of the final print in this screenshot is displayed below in the discussion section.

```
# the following loop uses the near_targets_to_sources dictionary to sum, per source building, the total
# number of occurances that source building has as being a specific target building's source. Thus,
# buildings with a higher corresponding integer are more prominant landmarks on the campus map

landmarks = dict()
for k, v in raw_data.items():
    name = v[0]
    if name in near_targets_to_sources.keys():
        for val in near_targets_to_sources[name]:
            #print(val)
            if val in landmarks.keys():
                landmarks[val] += 1
            else:
                landmarks[val] = 1

for k,v in landmarks.items():
    print(k, v)
```

```
SchapiroCEPSR 3
Mudd&EngTerrace&Fairchild&CS 4
NorthwestCorner 2
Uris 8
Pupin 2
Schermerhorn 4
Chandler&Havemeyer 4
Avery 2
Fayerweather 2
Mathematics 2
LowLibrary 6
StPaulChapel 3
EarlHall 2
Philosophy 2
Lewisohn 2
Dodge 2
Kent 2
Hamilton&Hartley&Wallach&JohnJay 4
CollegeWalk 4
Journalism&Furnald 4
Lerner 3
ButlerLibrary 4
Carman 2
```

```
# the following code block prints each building as a target on the left with its landmark source on the
# right (its source that is the source most often used as a source for other buildings also)

def get_num(raw_data, building_name):
    # takes in building name and returns its number
    for k,v in raw_data.items():
        if v[0] == building_name:
            return k

build_land = dict()
print("{:<32} {:<12}" .format('TARGET', 'LANDMARK'))

for key, val in raw_data.items():
    k = val[0]
    if k in near_targets_to_sources.keys():
        v = near_targets_to_sources[k]
        maxi = 0
        maxi_build = 'Carman'

        for x in v:
            if landmarks[x] > maxi:
                maxi = landmarks[x]
                maxi_build = x

        build_land[key] = get_num(raw_data, maxi_build)
        print("{:<32} {:<12}" .format(k, maxi_build))
    else:
        print("{:<32} {:<12}" .format(k, 'None'))
```

```
TARGET                           LANDMARK
Pupin                            Uris
SchapiroCEPSR                    Uris
Mudd&EngTerrace&Fairchild&CS     Uris
NorthwestCorner                  Chandler&Havemeyer
Uris                             Mudd&EngTerrace&Fairchild&CS
Schermerhorn                     Uris
Chandler&Havemeyer               Uris
OldComputerCenter                Uris
Avery                            Uris
Fayerweather                     Schermerhorn
Mathematics                      Chandler&Havemeyer
LowLibrary                       Uris
StPaulChapel                     LowLibrary
EarlHall                         LowLibrary
Lewisohn                         EarlHall
Philosophy                       Hamilton&Hartley&Wallach&JohnJay
Buell                            LowLibrary
AlmaMater                        LowLibrary
Dodge                            LowLibrary
Kent                             CollegeWalk
CollegeWalk                      Journalism&Furnald
Journalism&Furnald               CollegeWalk
Hamilton&Hartley&Wallach&JohnJay CollegeWalk
Lerner                           Journalism&Furnald
ButlerLibrary                    Journalism&Furnald
Carman                           ButlerLibrary
```

```
# build land is a dictionary where the keys are the target buildings' numbers and the corresponding value
# is the number of the most prominent landmark to that target building

print(build_land)
```

```
{9: 47, 19: 47, 28: 47, 38: 66, 47: 28, 57: 47, 66: 47, 76: 47, 85: 47, 94: 57, 104: 66, 113: 47, 123: 113, 132: 113,
142: 132, 151: 217, 161: 113, 170: 113, 179: 113, 189: 198, 198: 208, 208: 198, 217: 198, 236: 208, 246: 208, 255: 24
6}
```

Discussion:

For my nearTo(S, T) function that determines if target building T is near to source building S I decided to define nearness as T's MBR intersecting with the MBR of S that is twice the dimensions of S's original MBR. Thus, nearTo is affected by the dimensions of the source building S since the dimensions of the new, larger MBR of S are directly dependent on the

original width and height dimensions of S's original MBR (unless the original MBR is too close to the edges of the image, in this case the new MBR only expands into the directions where an MBR with twice the width and height of the original MBR can fit).

Nearness:

```
Pupin
S to T: SchapiroCEPSR NorthwestCorner
T to S: SchapiroCEPSR Mudd&EngTerrace&Fairchild&CS NorthwestCorner Uris

SchapiroCEPSR
S to T: Pupin Mudd&EngTerrace&Fairchild&CS Uris
T to S: Pupin Mudd&EngTerrace&Fairchild&CS Uris

Mudd&EngTerrace&Fairchild&CS
S to T: Pupin SchapiroCEPSR Uris Schermerhorn
T to S: SchapiroCEPSR Uris Schermerhorn

NorthwestCorner
S to T: Pupin Chandler&Havemeyer
T to S: Pupin Chandler&Havemeyer

Uris
S to T: Pupin SchapiroCEPSR Mudd&EngTerrace&Fairchild&CS Schermerhorn Chandler&Havemeyer OldComputerCenter Avery LowL
ibrary
T to S: SchapiroCEPSR Mudd&EngTerrace&Fairchild&CS Schermerhorn Chandler&Havemeyer

Schermerhorn
S to T: Mudd&EngTerrace&Fairchild&CS Uris Avery Fayerweather
T to S: Mudd&EngTerrace&Fairchild&CS Uris Avery Fayerweather

Chandler&Havemeyer
S to T: NorthwestCorner Uris OldComputerCenter Mathematics
T to S: NorthwestCorner Uris Mathematics

OldComputerCenter
S to T:
T to S: Uris Chandler&Havemeyer

Avery
S to T: Schermerhorn StPaulChapel
T to S: Uris Schermerhorn LowLibrary StPaulChapel

Fayerweather
S to T: Schermerhorn StPaulChapel
T to S: Schermerhorn StPaulChapel

Mathematics
S to T: Chandler&Havemeyer EarlHall
T to S: Chandler&Havemeyer EarlHall

LowLibrary
S to T: Avery StPaulChapel EarlHall Buell AlmaMater Dodge
T to S: Uris

StPaulChapel
S to T: Avery Fayerweather Philosophy
T to S: Avery Fayerweather LowLibrary Philosophy
```

```
EarlHall
S to T: Mathematics Lewisohn
T to S: Mathematics LowLibrary Lewisohn

Lewisohn
S to T: EarlHall Dodge
T to S: EarlHall Dodge

Philosophy
S to T: StPaulChapel Kent
T to S: StPaulChapel Kent Hamilton&Hartley&Wallach&JohnJay

Buell
S to T:
T to S: LowLibrary

AlmaMater
S to T:
T to S: LowLibrary

Dodge
S to T: Lewisohn CollegeWalk
T to S: LowLibrary Lewisohn CollegeWalk Journalism&Furnald

Kent
S to T: Philosophy CollegeWalk
T to S: Philosophy CollegeWalk Hamilton&Hartley&Wallach&JohnJay

CollegeWalk
S to T: Dodge Kent Journalism&Furnald Hamilton&Hartley&Wallach&JohnJay
T to S: Dodge Kent Journalism&Furnald Hamilton&Hartley&Wallach&JohnJay

Journalism&Furnald
S to T: Dodge CollegeWalk Lerner ButlerLibrary
T to S: CollegeWalk Lerner ButlerLibrary

Hamilton&Hartley&Wallach&JohnJay
S to T: Philosophy Kent CollegeWalk ButlerLibrary
T to S: CollegeWalk ButlerLibrary

Lerner
S to T: Journalism&Furnald ButlerLibrary Carman
T to S: Journalism&Furnald ButlerLibrary Carman

ButlerLibrary
S to T: Journalism&Furnald Hamilton&Hartley&Wallach&JohnJay Lerner Carman
T to S: Journalism&Furnald Hamilton&Hartley&Wallach&JohnJay Lerner Carman

Carman
S to T: Lerner ButlerLibrary
T to S: Lerner ButlerLibrary
```

The above table shows my nearTo relationships. The rows that begin with ‘S to T’ mean that the following buildings are the target buildings nearTo the source building that is the title of the section. For the ‘T to S’ rows the T input to nearTo is the title building of the section while the S inputs that result in a nearTo value of True are the following buildings in the row. The ‘S to T’ and ‘T to S’ rows of some buildings are not equivalent because the nearTo relation is only dependent on the dimensions of the S input. For example, Low Library’s larger MBR encloses Alma Mater but when Alma Mater is the source Low Library’s original MBR is not enclosed in Alma Mater’s larger MBR.

I decided to double the dimensions of the original MBR for my definition of nearby-ness, instead of using a different scalar, because conceptually it made sense that buildings within this imaginary bound of a larger rectangle with dimensions that are twice that of the original building MBR are near to the source building. When observing the outputs of this step the resulting ‘S to T’ and ‘T to S’ rows make sense for my definition of nearTo, thus I believe my design of this

step was successful. When determining whether original and larger building MBR's intersect, I used a similar algorithm to my intersect MBR function in Step 1.

Confusion:

The source building that is nearTo(S, T) the most target buildings is Uris with eight. The source building that is nearTo(S, T) the least target buildings is a tie between Alma Mater and Uris for zero. The target building that is nearTo(S, T) the most source buildings is a seven way tie between Pupin, Uris, Schermerhorn, Avery, StPaulChapel, Dodge, and ButlerLibrary, with 4 source buildings each. Finally, the target building that is nearTo(S, T) the least source buildings is a three way tie between AlmaMater, Buell, and LowLibrary, with one source building each.

Minimization:

The following table contains target buildings on the left with their corresponding landmark building on the right (the source building associated with target that is most commonly used as a source for other buildings too):

TARGET	LANDMARK
Pupin	Uris
SchapiroCEPSR	Uris
Mudd&EngTerrace&Fairchild&CS	Uris
NorthwestCorner	Chandler&Havemeyer
Uris	Mudd&EngTerrace&Fairchild&CS
Schermerhorn	Uris
Chandler&Havemeyer	Uris
OldComputerCenter	Uris
Avery	Uris
Fayerweather	Schermerhorn
Mathematics	Chandler&Havemeyer
LowLibrary	Uris
StPaulChapel	LowLibrary
EarlHall	LowLibrary
Lewisohn	EarlHall
Philosophy	Hamilton&Hartley&Wallach&JohnJay
Buell	LowLibrary
AlmaMater	LowLibrary
Dodge	LowLibrary
Kent	CollegeWalk
CollegeWalk	Journalism&Furnald
Journalism&Furnald	CollegeWalk
Hamilton&Hartley&Wallach&JohnJay	CollegeWalk
Lerner	Journalism&Furnald
ButlerLibrary	Journalism&Furnald
Carman	ButlerLibrary

It makes sense that Uris is the most common since it is one of the largest buildings in one of the most populated areas of the map. The other landmark buildings also make sense based on their size and locations, so the algorithms of this step were successful. In its entirety, this step outputted a 100% success rate.

Step 5

Code:

```
def what_and_where(raw_data, building_num):
    si = size_(raw_data, building_num)
    ar = aspectRatio(raw_data, building_num)
    geo = geometry(raw_data, building_num)
    vloc = vertical_location(raw_data, building_num, tot_height)
    hloc = horizontal_location(raw_data, building_num, tot_width)
    ori = orientation(raw_data, building_num)

    all_info[raw_data[building_num][0]] = [si,ar,geo,vloc,hloc,ori]

    return "{} {} {} {} {} {} building".format(si,ar,geo,vloc,hloc,ori)

def description(raw_data, building_num):
    print("The " + what_and_where(raw_data, building_num), end = " ")
    print("near the " + what_and_where(raw_data, build_land[building_num]))


all_info = dict()

for key, val in raw_data.items():
    print(val[0] + " (near landmark {}):".format(raw_data[build_land[key]][0]))
    description(raw_data, key)
    print()
```

```
# the following code determines which buildings have all-description confusion

#all_confusion = dict()

for k1,v1 in all_info.items():
    for k2,v2 in all_info.items():
        if k1 == k2:
            continue

        if v1 == v2:
            print(k1, k2)
#             if k1 in all_confusion.keys():
#                 all_confusion[k1].append(k2)
#             else:
#                 all_confusion[k1] = [k2]

#             if k2 in all_confusion.keys():
#                 all_confusion[k2].append(k1)
#             else:
#                 all_confusion[k2] = [k1]

# for k,v in all_confusion.items():
#     print(k,v)
```

```
Mathematics Lewisohn
Lewisohn Mathematics
```

Discussion:

Total Description:

The following table displays the result of my system's natural language sentences describing each building in their entirety, which includes each descriptor from each step.

Name	Long Description
9 Pupin	The Small Wide Asymmetrical Uppermost Left Horizontally-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building
19 SchapiroCEPSR	The Tiny Medium-Width Square-shaped Upper Mid-Width Horizontally-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building
28 Mudd&EngTerrace &Fairchild&CS	The Large Wide C-shaped Upper Right Horizontally-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building
38 NorthwestCorner	The Small Narrowest Rectangle-shaped Upper Leftmost Vertically-Oriented building near the Medium Medium-Width Square-shaped Upper Left Horizontally-Oriented building
47 Uris	The Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building near the Large Wide C-shaped Upper Right Horizontally-Oriented building
57 Schermerhorn	The Medium Wide Asymmetrical Upper Right Horizontally-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building
66 Chandler&Haveme yer	The Medium Medium-Width Square-shaped Upper Left Horizontally-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building

76 OldComputerCenter	The Minuscule Narrow Rectangle-shaped Upper Mid-Width Vertically-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building
85 Avery	The Tiny Narrow Asymmetrical Mid-Height Right Vertically-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building
94 Fayerweather	The Tiny Narrow Rectangle-shaped Mid-Height Rightmost Vertically-Oriented building near the Medium Wide Asymmetrical Upper Right Horizontally-Oriented building
104 Mathematics	The Tiny Narrow I-shaped Mid-Height Left Vertically-Oriented building near the Medium Medium-Width Square-shaped Upper Left Horizontally-Oriented building
113 LowLibrary	The Medium Medium-Width Square-shaped Mid-Height Mid-Width Vertically-Oriented building near the Large Narrow C-shaped Upper Mid-Width Vertically-Oriented building
123 StPaulChapel	The Tiny Wide Asymmetrical Mid-Height Right Horizontally-Oriented building near the Medium Medium-Width Square-shaped Mid-Height Mid-Width Vertically-Oriented building
132 EarlHall	The Tiny Wide Rectangle-shaped Mid-Height Left Horizontally-Oriented building near the Medium Medium-Width Square-shaped Mid-Height Mid-Width Vertically-Oriented building
142 Lewisohn	The Tiny Narrow I-shaped Mid-Height Left Vertically-Oriented building near the Tiny Wide Rectangle-shaped Mid-Height Left Horizontally-Oriented building
151 Philosophy	The Tiny Narrow I-shaped Mid-Height Right Vertically-Oriented building near the Largest Narrow Asymmetrical Lower Right Vertically-Oriented building
161 Buell	The Minuscule Wide C-shaped Mid-Height Right Horizontally-Oriented building near the Medium Medium-Width Square-shaped Mid-Height Mid-Width Vertically-Oriented building

170 AlmaMater	The Smallest Medium-Width Square-shaped Mid-Height Mid-Width Non-Oriented building near the Medium Medium-Width Square-shaped Mid-Height Mid-Width Vertically-Oriented building
179 Dodge	The Small Wide C-shaped Mid-Height Left Horizontally-Oriented building near the Medium Medium-Width Square-shaped Mid-Height Mid-Width Vertically-Oriented building
189 Kent	The Small Wide Asymmetrical Mid-Height Right Horizontally-Oriented building near the Gigantic Widest Rectangle-shaped Mid-Height Mid-Width Horizontally-Oriented building
198 CollegeWalk	The Gigantic Widest Rectangle-shaped Mid-Height Mid-Width Horizontally-Oriented building near the Medium Medium-Width Square-shaped Lower Left Horizontally-Oriented building
208 Journalism&Furnald	The Medium Medium-Width Square-shaped Lower Left Horizontally-Oriented building near the Gigantic Widest Rectangle-shaped Mid-Height Mid-Width Horizontally-Oriented building
217 Hamilton&Hartley&Wallach&JohnJay	The Largest Narrow Asymmetrical Lower Right Vertically-Oriented building near the Gigantic Widest Rectangle-shaped Mid-Height Mid-Width Horizontally-Oriented building
236 Lerner	The Medium Wide Rectangle-shaped Lower Left Horizontally-Oriented building near the Medium Medium-Width Square-shaped Lower Left Horizontally-Oriented building
246 ButlerLibrary	The Large Wide Rectangle-shaped Lower Mid-Width Horizontally-Oriented building near the Medium Medium-Width Square-shaped Lower Left Horizontally-Oriented building
255 Carman	The Small Wide Rectangle-shaped Lowermost Left Horizontally-Oriented building near the Large Wide Rectangle-shaped Lower Mid-Width Horizontally-Oriented building

Confusion and Minimization:

According to my program, the only buildings that still contain all-description confusion are Mathematics and Lewisohn. This makes sense because they are two T-shaped, similar-sized buildings in the same part of campus. In order to minimize this confusion, one can refer to Mathematics as the upper of the two buildings, as follows:

Mathematics minimization: “The Upper Tiny Narrow I-shaped Mid-Height Left Vertically-Oriented building near the Medium Medium-Width Square-shaped Upper Left Horizontally-Oriented building.”
Lewisohn minimization: “The Lower Tiny Narrow I-shaped Mid-Height Left Vertically-Oriented building near the Medium Medium-Width Square-shaped Upper Left Horizontally-Oriented building.”

Overall, the smallest unambiguous description of a building is “Upper Gigantic” which refers to Uris and the longest is “The Upper Tiny Narrow I-shaped Mid-Height Left Vertically-Oriented building near the Medium Medium-Width Square-shaped Upper Left Horizontally-Oriented building” which refers to Mathematics. I know Uris has the smallest unambiguous description through observation and counting by hand that was done by comparing the descriptions of each building on campus.

The all-description output of my algorithm was almost 100% accurate overall. The only element that caused a non-perfect performance was the fault in geometry detection discussed earlier. In general, a user of this system would be able to find their way around campus which I believe makes it a success.