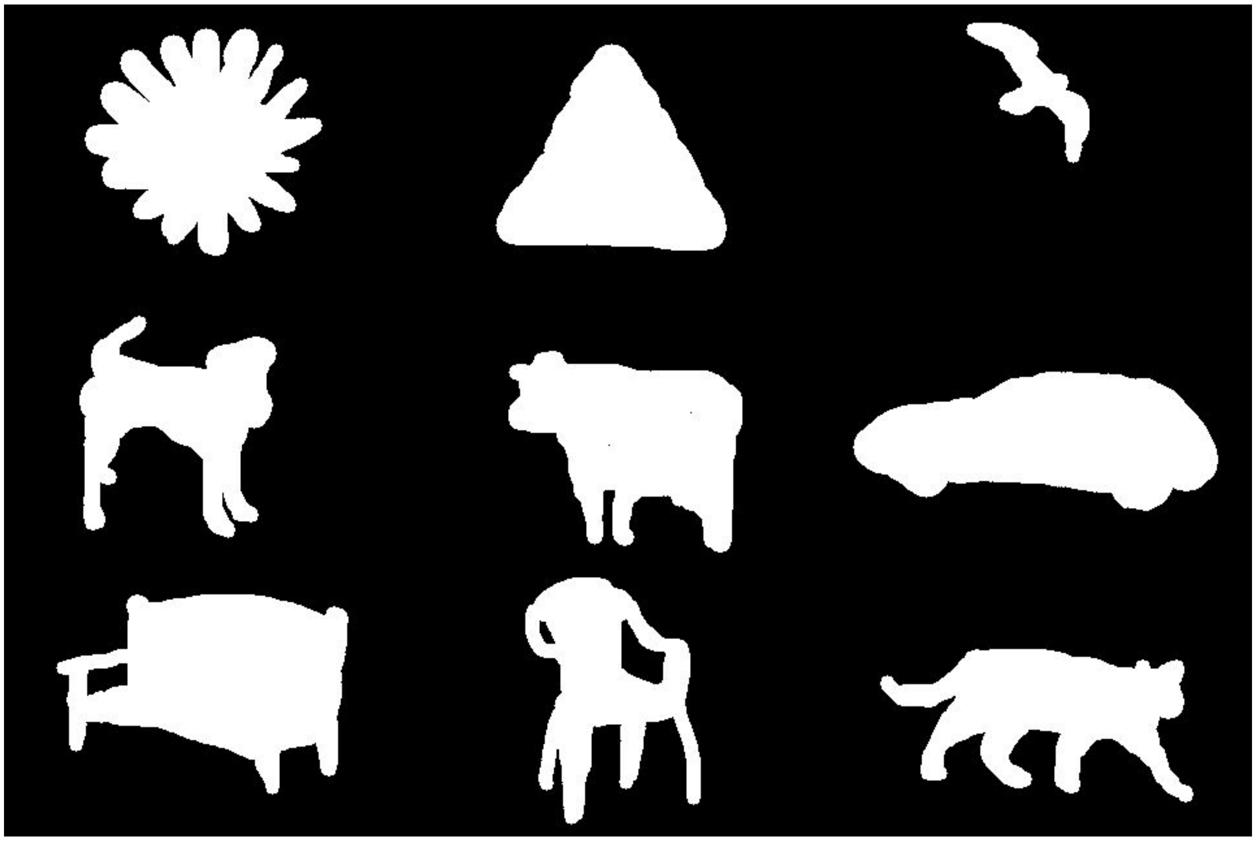
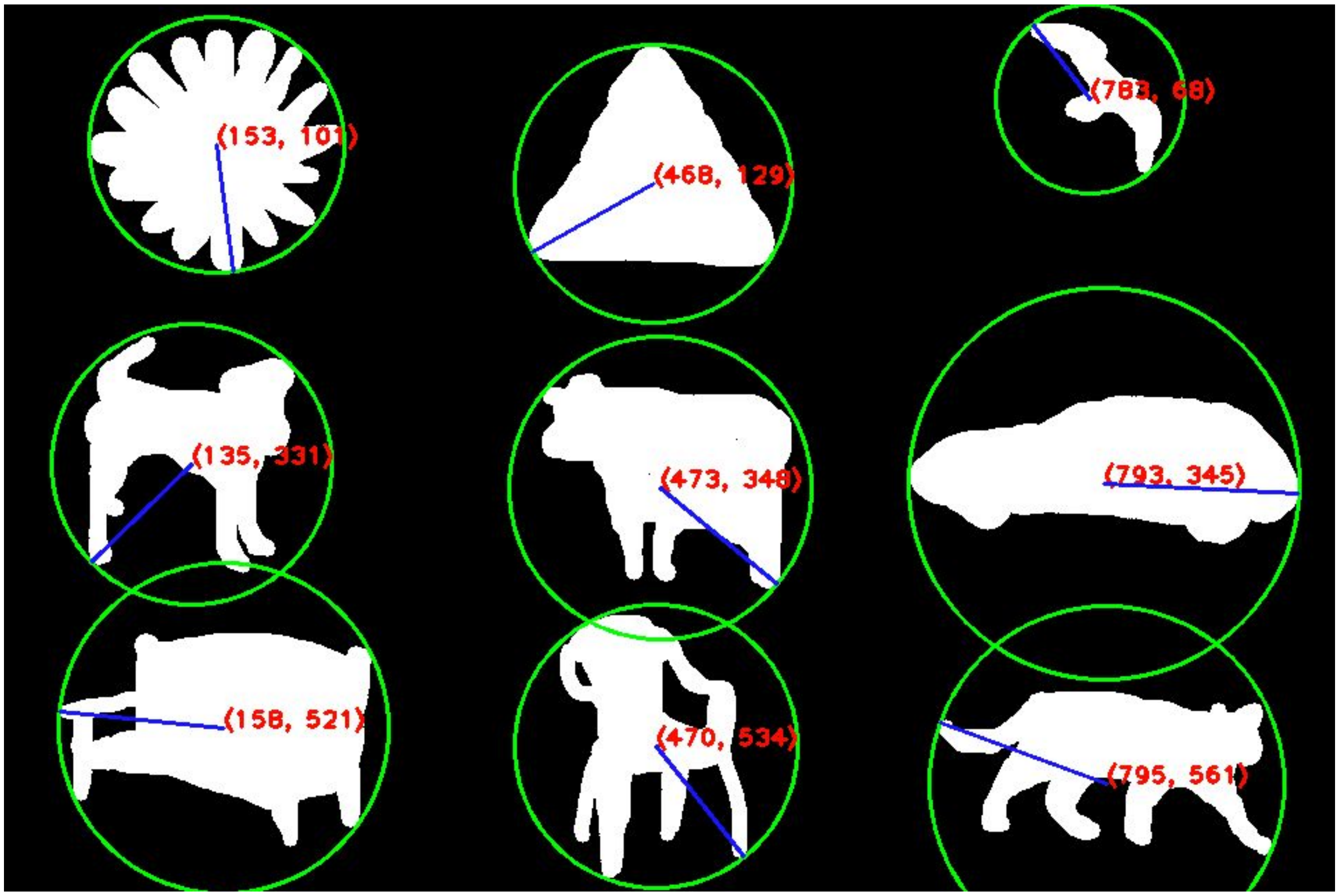
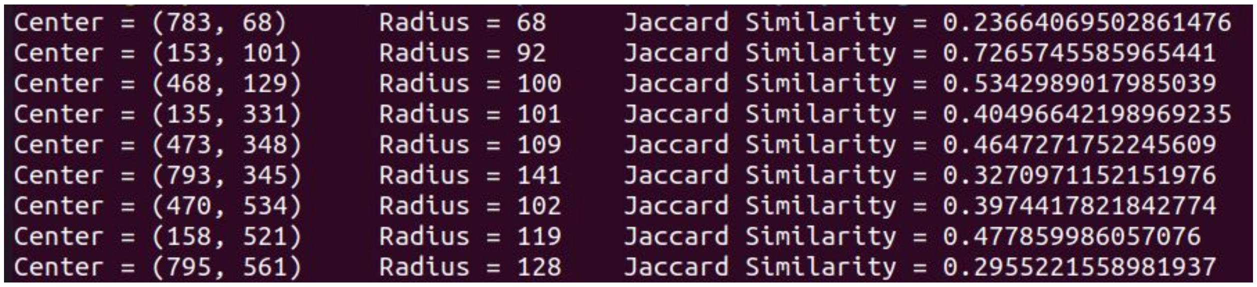
A1-report

Input 1



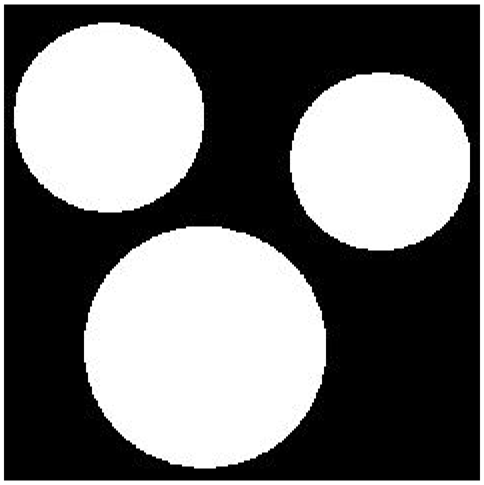
Output 1



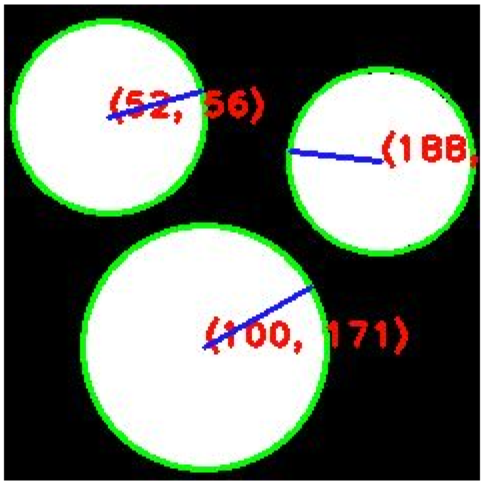


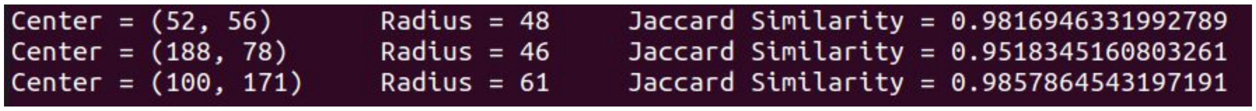
```
Center = (783, 68)      Radius = 68     Jaccard Similarity = 0.23664069502861476
Center = (153, 101)     Radius = 92     Jaccard Similarity = 0.7265745585965441
Center = (468, 129)     Radius = 100    Jaccard Similarity = 0.5342989017985039
Center = (135, 331)     Radius = 101    Jaccard Similarity = 0.40496642198969235
Center = (473, 348)     Radius = 109    Jaccard Similarity = 0.4647271752245609
Center = (793, 345)     Radius = 141    Jaccard Similarity = 0.3270971152151976
Center = (470, 534)     Radius = 102    Jaccard Similarity = 0.3974417821842774
Center = (158, 521)     Radius = 119    Jaccard Similarity = 0.477859986057076
Center = (795, 561)     Radius = 128    Jaccard Similarity = 0.2955221558981937
```

Input 2



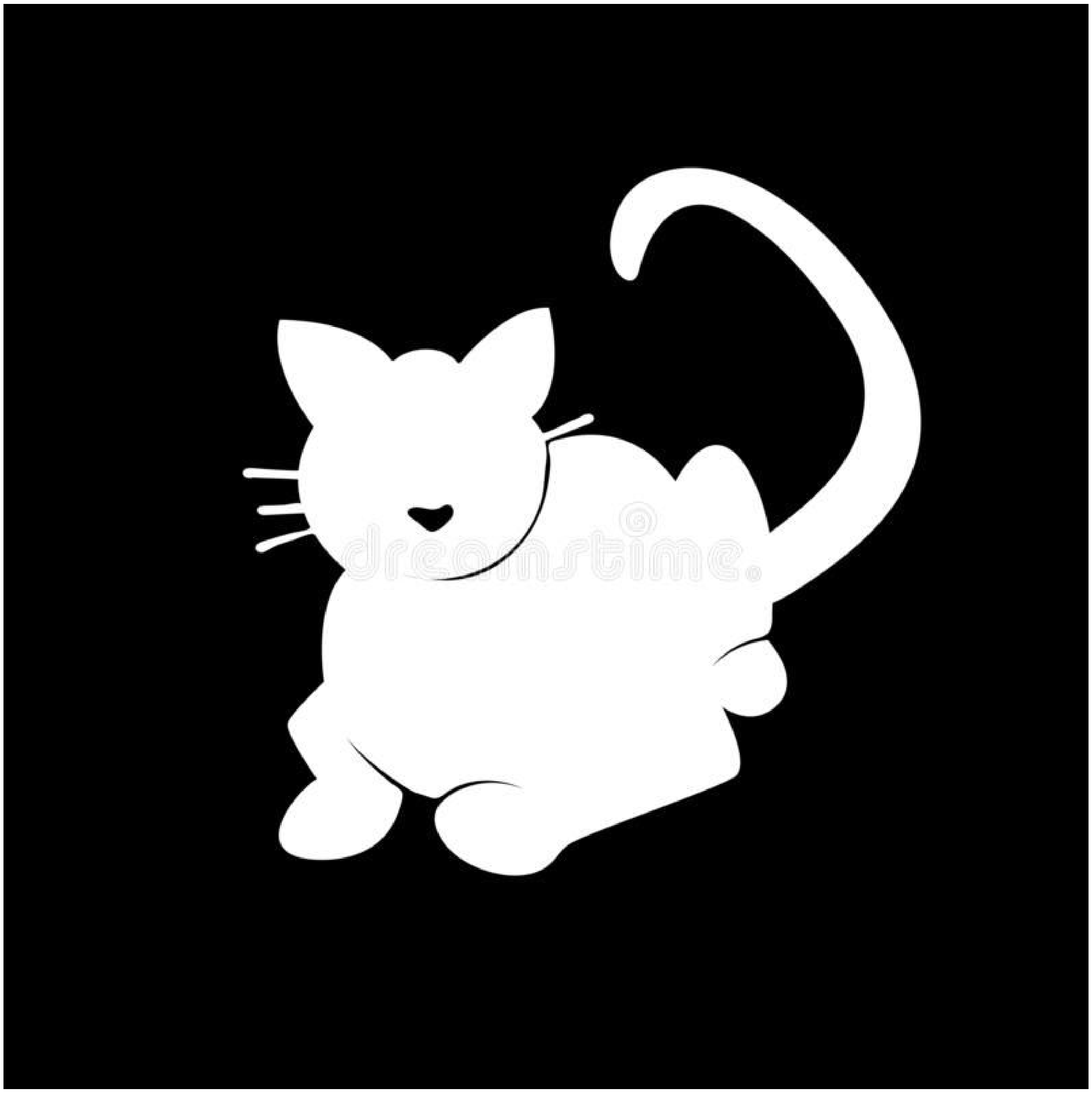
Output 2



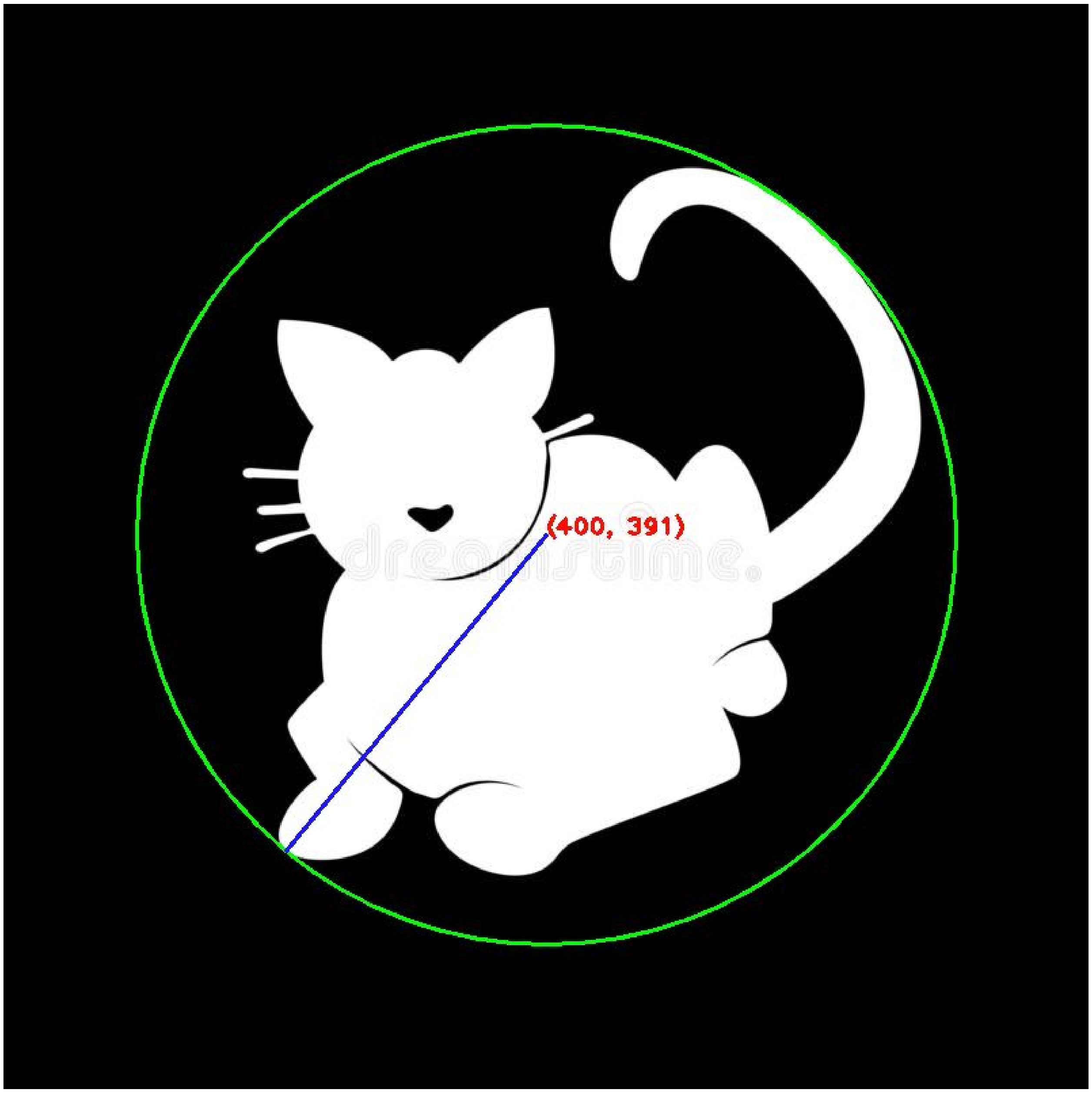


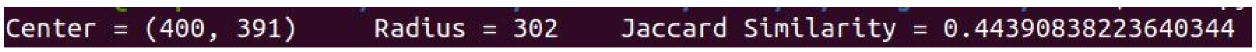
```
Center = (52, 56)       Radius = 48     Jaccard Similarity = 0.9816946331992789
Center = (188, 78)      Radius = 46     Jaccard Similarity = 0.9518345160803261
Center = (100, 171)     Radius = 61     Jaccard Similarity = 0.9857864543197191
```

Input 3



Output 3




```
Center = (400, 391)     Radius = 302    Jaccard Similarity = 0.44390838223640344
```

Methodology for finding bounding circle:

1. Find the diagonal coordinates of the tightest bounding rectangle and consider its center.
2. There is a parameter **search_length** whose value can be passed from the command line.
3. This parameter will make a square around the center and all the pixels inside it will be considered as center one by one and we will find the farthest point of the object from the current center.
4. We will consider the final center of the bounding circle as the one which has the minimum farthest point from the center.

Comments on the methodology:

- Increasing the value of parameter **search_length** will improve the results but at the same time, the execution time will also increase though not drastically.
- Usually, **search_length** = 5 gives pretty good results.

Following stats on the image attached in the assignment on the google classroom.

search_length	**Execution Time(in sec)**
1	1.05
5	1.10
10	1.22
50	2.41
100	4.56

Optimization(s):

- In step-3, only those points which are at the border of the object are chosen for finding the farthest point as inside points do not matter.
- Border points are found while finding connected components, thereby saving the second iteration.
- Early breaking when the farthest point is already greater than our current radius is used and still we need to find the distance for border points left.

Methodology for Jaccard Similarity:

- The number of pixels making the object is found while finding the connected components.
- In the Jaccard similarity function, I passed the center and radius of the circle of an object and count the number of pixels present inside that circle.
- **Similarity = number of pixels making the object/number of pixels inside the circle.**

Code

```python
import cv2
import sys
from queue import Queue
from math import *

white = 255
black = 0
object_color = white
background_color = black
bounding_circle_info = {}
search_length = 5


def color_to_BW(color_image, threshold=127):
    '''
    converts colored image to black and white
    '''
    grayImage = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)
    thresh, bw_image = cv2.threshold(grayImage, threshold, white, cv2.THRESH_BINARY)
    return bw_image


def is_valid_move(x, y, rows, cols):
    '''
    check validity of cell whether it is inside matrix or not
    '''
    return x < rows and y < cols and x >= 0 and y >= 0


def color_the_component(row, col, color, bw_image):
    '''
    main work: color the given component with color using BFS
    side work: find coordinates which are at perimeter of object
                and diagonal coordinates of bounding rectangle
    '''
    rows, cols = len(bw_image), len(bw_image[0])
    # below corrdinates are diagonals of bounding rectangle
    x_min, y_min = col, row
    x_max, y_max = col, row
    # store all coordinates which are at perimeter of current object
    border_x_y = set([(col, row)])
    num_white_pixels = 1
    q = Queue()

    q.put((row, col))
     # color current pixel
    bw_image[row][col] = color

    # Note: 8 nearby pixels are possible
    # but we are taking only in 4 directions
    # as single diagonal of object will not be possible to see with naked eyes
    moves = [(0, 1), (1, 0), (-1, 0), (0, -1)]

    while q.qsize() != 0:

       x1, y1 = q.get()

       # check nearby pixels
       for move in moves:
             x2, y2 = x1 + move[0], y1 + move[1]

             if is_valid_move(x2, y2, rows, cols) and (bw_image[x2][y2] == object_color):
                   # nearby pixel is valid and is of white color then color that nearby
pixel

                   q.put((x2, y2)) # keep nearby pixels in the queue for further recursive
coloring
                   bw_image[x2][y2] = color # color nearby pixel
                   num_white_pixels += 1


                   # find whether (x2, y2) are at perimeter of current object
                   for _move in moves:
                         x3, y3 = x2 + _move[0], y2 + _move[1]
                         if not is_valid_move(x3, y3, rows, cols) or bw_image[x3][y3] ==
background_color:
                               border_x_y.add((y2, x2))
                               # find diagonal points of bounding rectangle
                               x_min = min(x_min, y2)
                               x_max = max(x_max, y2)
                               y_min = min(y_min, x2)
                               y_max = max(y_max, x2)
                               break
    find_bounding_circle_info(x_min, x_max, y_min, y_max, border_x_y, rows, cols, color,
num_white_pixels)


def find_bounding_circle_info(x_min, x_max, y_min, y_max, border_x_y, rows, cols, color,
num_white_pixels):
    '''
    This will find center, farthest point from center and radius of circle
    and store it in bounding_circle_info with its corresponding color
    '''
    center = ((x_min + x_max) >> 1, (y_min + y_max) >> 1)
    farthest_point = (0, 0)
    min_radius = 10**18

    for center_x in range(max(0, center[0] - search_length), min(cols, center[0] +
search_length)):
       for center_y in range(max(0, center[1] - search_length), min(rows, center[1] +
search_length)):

             # find radius (center_x, center_y) are assumed to be center of bounding
circle
             max_radius = 0
             local_farthest_point = (0, 0)
             for x, y in border_x_y:
                   r = (x - center_x) ** 2 + (y - center_y) ** 2

                   if r > max_radius:
                         max_radius = r
                         local_farthest_point = (x, y)

                         if max_radius >= min_radius:
                               break

             if min_radius > max_radius:
                   min_radius = max_radius
                   farthest_point = local_farthest_point
                   center = (center_x, center_y)

    bounding_circle_info[abs(color)] = (center, farthest_point, ceil(min_radius**0.5),
num_white_pixels)


def find_connected_components(bw_image):
    '''
    find all connected components formed by objects in a binary image
    '''
    rows, cols = bw_image.shape
    bw_image = bw_image.tolist() # numpy matrix to normal list of lists for fast access

    color = 0

    for row in range(rows):
       for col in range(cols):
             if bw_image[row][col] == object_color:
                   color -= 1
                   color_the_component(row, col, color, bw_image)


def make_bounding_circle(originalImage):
    rows, cols = originalImage.shape[:2]
    for i in bounding_circle_info:
       center, farthest_point, radius, num_white_pixels = bounding_circle_info[i]
       # ignore too small objects
       if radius > 10:
             jaccard_similarity = find_jaccard_similarity(rows, cols, center, radius,
num_white_pixels)
             print("Center =", center, "\tRadius =", radius, "\tJaccard Similarity =",
jaccard_similarity)

             originalImage = cv2.circle(originalImage, center, radius, (0, 255, 1), 2) #
draw circle
             cv2.putText(originalImage, str(center), center, cv2.FONT_HERSHEY_PLAIN, 1.2,
(0, 12, 255), 2) # put cordinates of center
             cv2.line(originalImage ,center, farthest_point, (255, 21, 25), 2) # make a
line from center to farthest point

    cv2.imshow('Ojects detected', originalImage)
    cv2.waitKey()
    cv2.destroyAllWindows()


def find_jaccard_similarity(rows, cols, center, radius, num_white_pixels):
    num_pixels_in_circle = 0
    sq_radius = radius ** 2

    for x in range(center[1] - radius, center[1] + radius):
```

```python
       for y in range(center[0] - radius, center[0] + radius):
             distance_from_center = (y- center[0])**2 + (x - center[1])**2
             if distance_from_center <= sq_radius:
                   num_pixels_in_circle += 1

    return num_white_pixels / num_pixels_in_circle


def main(image_path, _search_length=1):
    global search_length
    search_length = _search_length

    originalImage = cv2.imread(image_path)
    bw_image = color_to_BW(originalImage) # convert color to BW image
    find_connected_components(bw_image)
    make_bounding_circle(originalImage)



if __name__ == "__main__":

    if len(sys.argv) != 3:
       print("Usage: python3 assn1.py <path_of_image> <search_length>")
       print("\nsearch_length : a positive integer")
       print("\t\thigher its value --> higher accuracy of center of bounding circle")
       print("\t\tat the same time, execution time also increases")
       print("\t\t5 is usually found to be good estimate with execution time less than 2
seconds")
       exit(1)

    image_path = sys.argv[1]
    _search_length = max(1, int(sys.argv[2]))
    main(image_path, _search_length)
```