EE569 HW#3

Student Name: Shi-Lin Chen
Student ID: 2991911997
Student Email: shilinch@usc.edu
Submission Date: Mar 03, 2020

Problem 1: Geometric Image Modification

1.1 Motivation and Abstract

In this problem, we will apply different geometric modification, including spatial warping techniques and homographic transformation and image stitching. In (a.) we will learn different geometric operations such as: translation, scaling, rotation and their combinations in order to make some interesting image warping. In (b.) we will learn image stitching, including finding matching point pairs then warp. Through this problem, we could have better understanding about basic image modification.

1.2 Approach and Result

1.2(a.) Geometric warping

Warping Part:

This problem asks us to warp a square image into a circle. We could choose different ways to warp the image if our method could satisfy 3 requires, which are boundary, center and reversible mapping. I will use some drawing below to explain my method when implementing warping.

Basically, my method is to calculate the length of **Chords** (which I named bar) in the 1/4 circle. We already knew that the radius of the circle is 512/2 = 256. Therefore, I calculated 256 chords lengths in the 1/4 circle. With radius and chord lengths, we could use **Scaling** to implement the warping.

In the middle graph below, we could see there are **green** and **orange** lines. Each green + orange is length of 256 pixels, orange lines are the chords length I calculated (which the method is provided below). With these green and orange lines, we could warp the image row by row by scaling!

```
for(double i = 0; i < 256; i++) {
    degree = asin(i/256);
    bar.push_back(256*cos(degree));
```

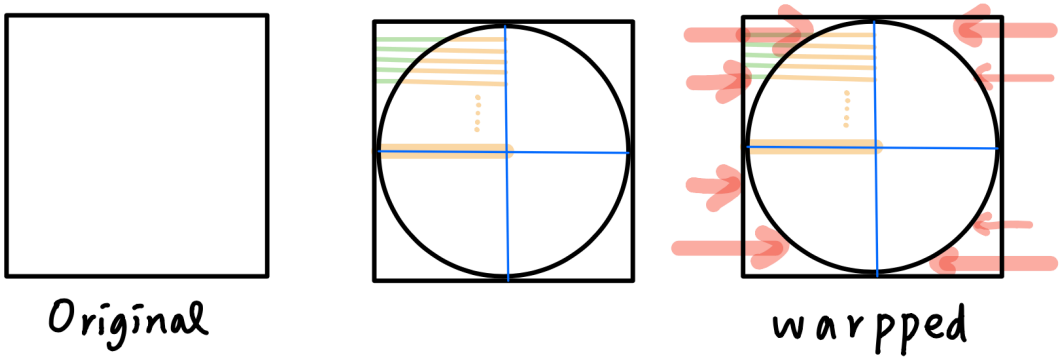
Original (green + orange) -> orange for each row. Since the directions are different, we should do left and right circle separate. Image below is 1/4 circle of warping, and other 3/4 circle are using the same method.

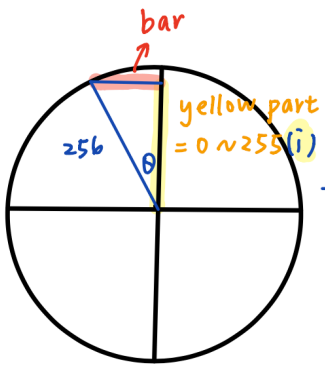
Scaling

- $T = \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_1 x \\ s_2 y \\ 1 \end{bmatrix}$

```
NewCol_2 = col * bar[i] / 256;

// Right-up Circle
ImagedataWarp[row][NewCol_2+256][0] = Imagedata[row][col+256][0];
ImagedataWarp[row][NewCol_2+256][1] = Imagedata[row][col+256][1];
ImagedataWarp[row][NewCol_2+256][2] = Imagedata[row][col+256][2];
```





from left graph,

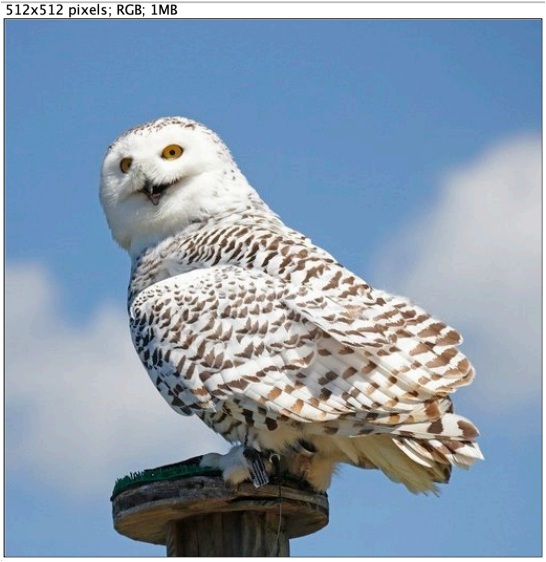
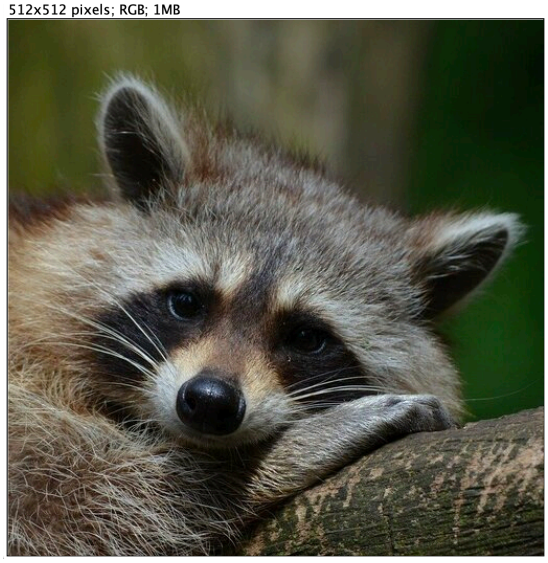
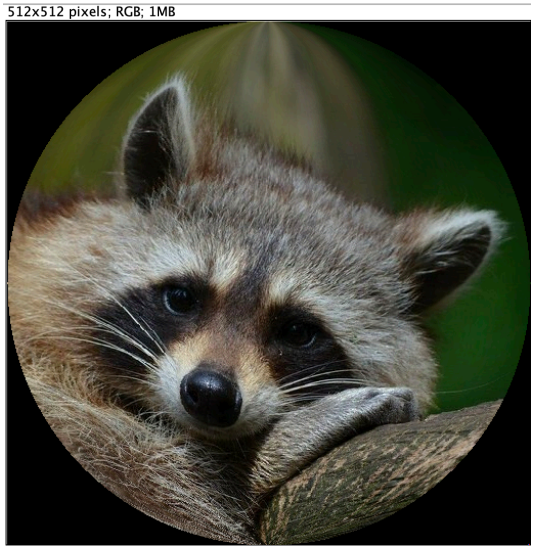
bar = $\sin\theta \cdot 256$

we can get θ by knowing radius & i

→ $\cos\theta = \frac{i}{256}$, $\theta = a\cos\left(\frac{i}{256}\right)$

∴ bar = $\sin\left(a\cos\left(\frac{i}{256}\right)\right) \cdot 256$ for $i = 0 \sim 255$

Result:

Original Input		Warp Imaged
512x512 pixels; RGB; 1MB 	>	512x512 pixels; RGB; 1MB 
512x512 pixels; RGB; 1MB 	>	512x512 pixels; RGB; 1MB 

Reverse Part:

Since I use scaling method to implement warping, I use the same method to reverse to the original Image. But this time, the Reverse column is at the warped image.

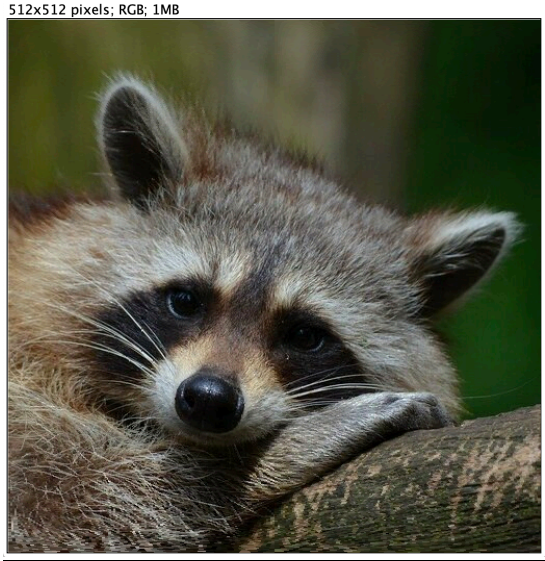
By this method, we could simply reverse the warping image to the original image (with a little bit artifact, which I will explain and compare in the discussion part.)

```
ReverseCol_2 = col * bar[i] / 256;

// Right-up Circle
ImagedataReverse[row][col+256][0] = ImagedataWarp[row][ReverseCol_2+256][0];
ImagedataReverse[row][col+256][1] = ImagedataWarp[row][ReverseCol_2+256][1];
ImagedataReverse[row][col+256][2] = ImagedataWarp[row][ReverseCol_2+256][2];
```

Reverse Result:

512x512 pixels; RGB; 1MB

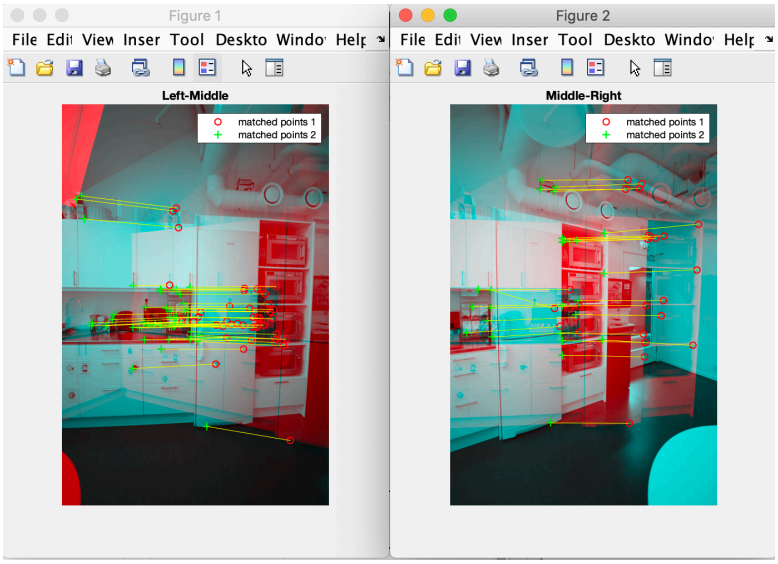


512x512 pixels; RGB; 1MB

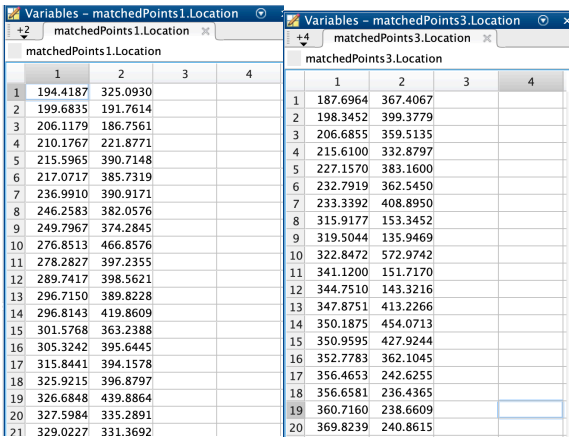
Although there are some artifacts of the reverse image, there are not miss pixels (black dots) in my reverse results. Thus, I didn't use linear interpolation to recover the pixels. Comparing result will be provided in my discussion part below.

1.3(b.) Homographic Transformation and Image Stitching

In this part, we have several steps to implement Image stitching. First, we could use SURF (Speed Up Robust Features) or SIFT(Scale-Invariant Feature Transform) to detect the features on different Images. Then we could match these matched features and the find the matched point pairs of two images, in this problem, we need to find matched point pairs of (left-middle) and (middle-right).



(Source: https://www.mathworks.com/help/vision/ref/matchfeatures.html)



Part of matching points of left-middle and middle-right image.

Then we need to find the good matched point pairs to calculate our homographic matrix.

Assume pair(x,y) belong to middle image, (u,v) belong to left or right image.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ w \end{pmatrix}$$ where x = x'/w and y = y'/w. Since we are taking 4 points, would get 8 equations then we could solve the value of a – h.

$$\begin{pmatrix} u_0 & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 0 & u_0 & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 0 & u_3 & v_3 & 1 & -u_3y_3 & -v_3y_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

After solving the linear equation and get the homographic matrix, we could use this matrix to do our warping. Before doing warping, we also need to create a large enough "canvas" to hold 3 images (the final result). Final result is to warp 3 images together into this large "canvas".

Result:

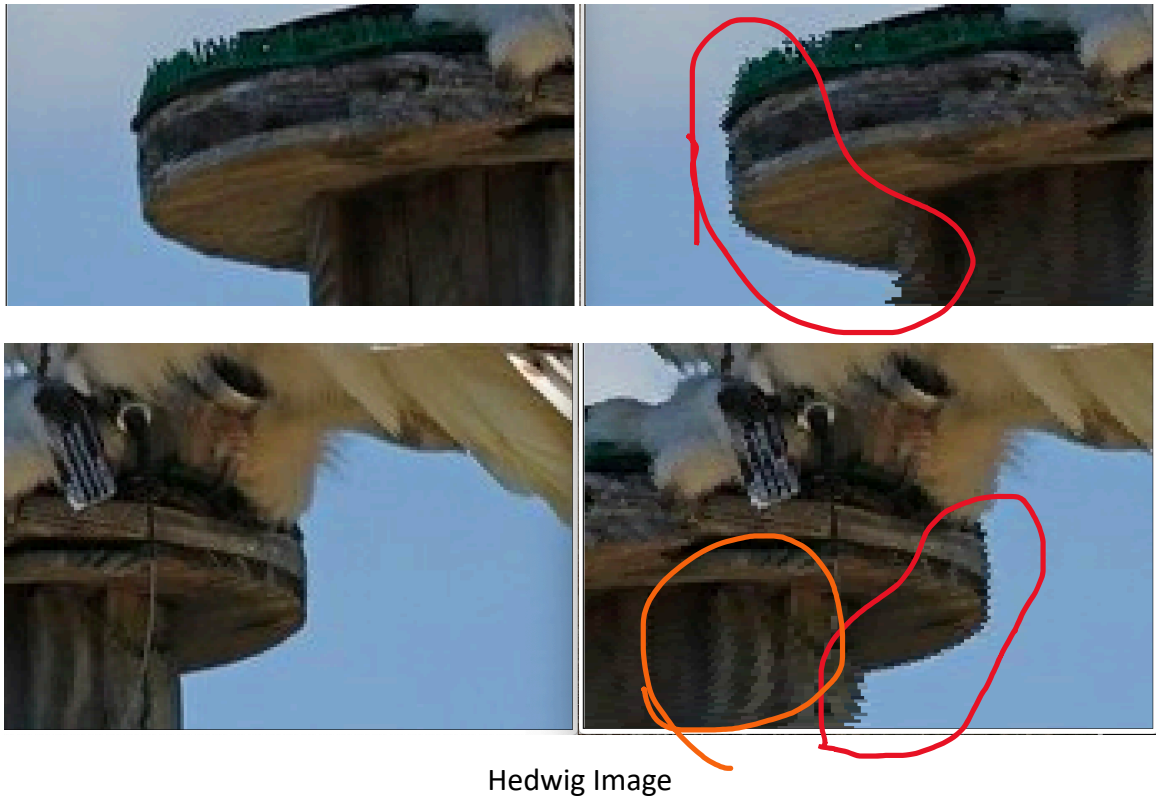
Due to the lack the of time, I have not successfully created the image of this problem. But I have wrote down the method to implement it and also the discussion part below.

1.3 Discussion

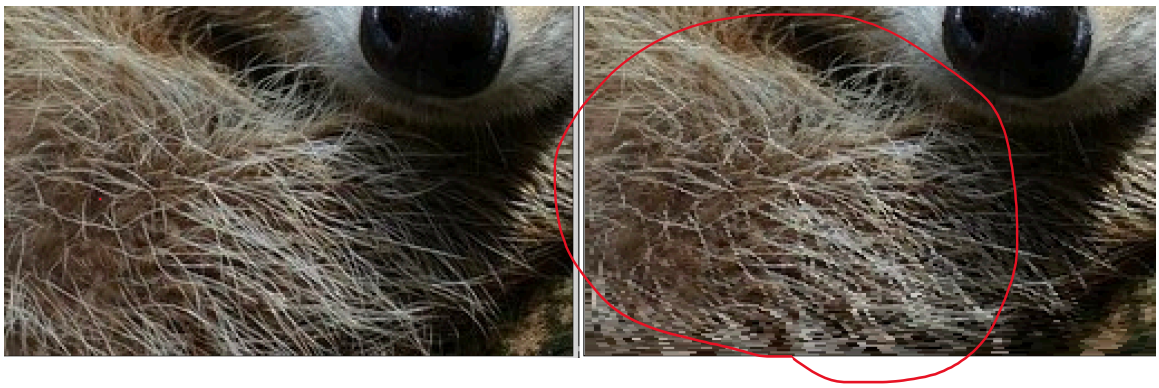
1.3(a.) Geometric Warping

1. Approach has been discussed in **1.2 approach part.**
2. In the last part.
3. Compare between **original image** and **reverse image**:

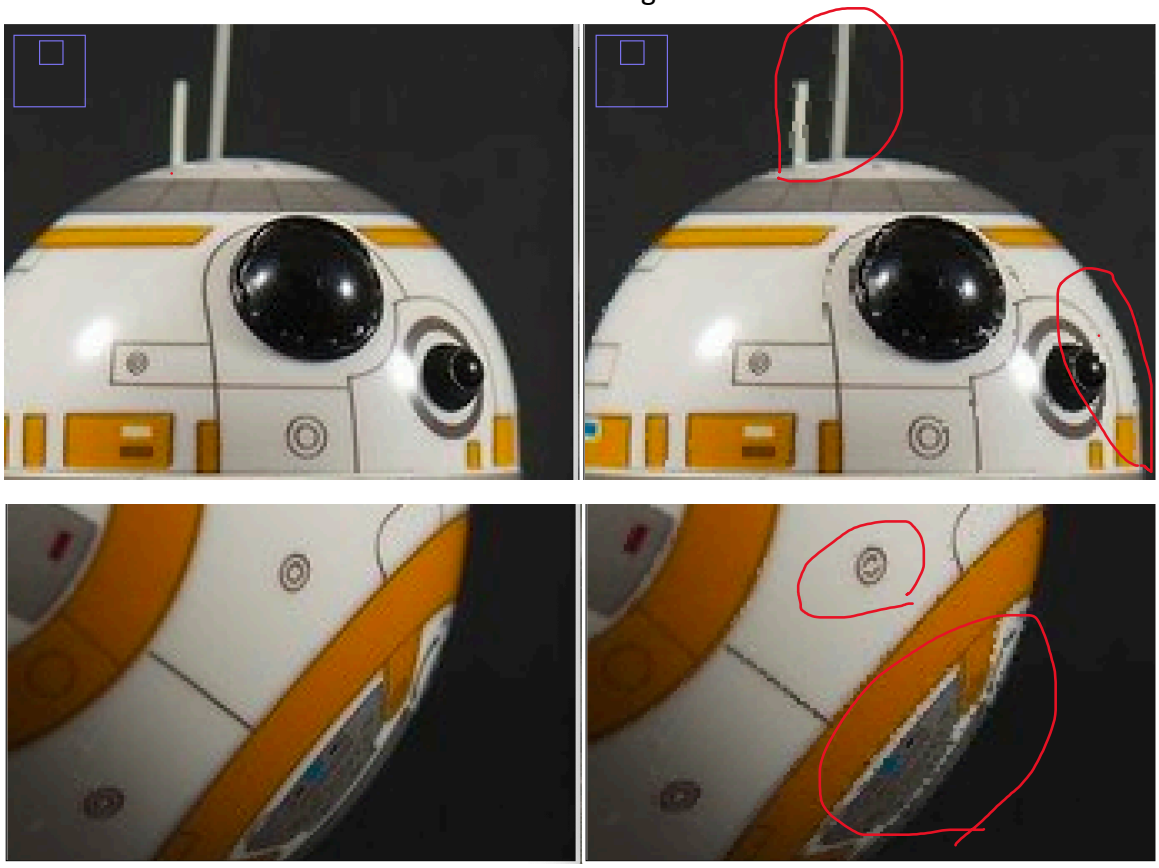
Original Image **Reverse Image**



Hedwig Image



Racoon Image



Bb8 Image

By zooming in the result, we could observe that there are some artifacts in the reversing images. In my implementation, I didn't lose a lot of pixels (black dots) in reversing part. Instead of black dots, there are many "**blur boundaries"** in many parts of my reverse image which could simply be seen in zoom-in image.

I think the reason of distortion is from the "**lose of pixels**". Since when we implement warping image, we will squeeze the image from square to circle. That is, the original amount of pixels with information are squeezed in the smaller region (less amount of pixels with information.)

Therefore, when we implement the reversing part, smaller region needed to be recovered to original large region. It means that less amount of pixels needed to be mapped into original amount of pixels. This procedure causes the distortion of the reversing image!

1.3(b.) Homographic Transformation and Image Stitching

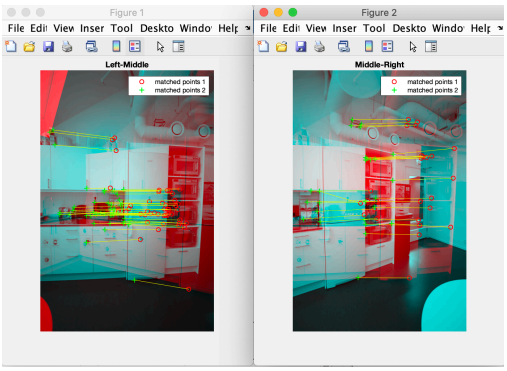
1. If two images needed to be stitched, we need 4 control points pair. There are Left-Middle and Right-Middle needed to be stitched, therefore, we need total 8 control points pairs.

1	194.4187	325.0930	1	129.3669	326.2923
2	199.6835	191.7614	2	26.8401	170.6606
3	206.1179	186.7561	3	36.1134	164.9852
4	210.1767	221.8771	4	40.6991	206.3105

- for Left-Middle

1	187.6964	367.4067	1	73.6721	333.6243
2	198.3452	399.3779	2	30.2598	410.6071
3	206.6855	359.5135	3	40.1623	363.5623
4	215.6100	332.8797	4	51.2582	333.2838

- Middle-Right



2. I think one way to decide which control points should be selected is their distance in the image. For example, I will choose control points that are not very close to each other, choosing close control points might cause the bad performance when implement stitching.

Problem 2: Morphological Processing

2.1 Motivation and Abstract

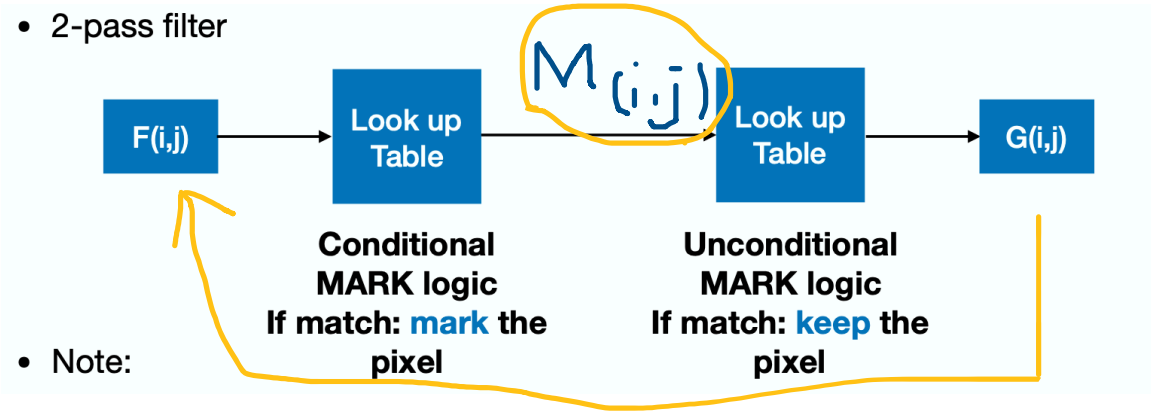
In this problem, we will learn basic morphological process implementation like Shrinking,

Thinning, and Skeleton in the first question. In following questions, we will apply morphological processing to implement some interesting staffs, such as counting objects in a binary image.

2.2 Approach and Result

2.2(a.) Basic morphological process implementation

In this problem, we need to binarize the image pixels value into 1 and 0, in order to match with the conditional and unconditional look up table (LUT). Since the given images are not only 0 and 255 (there are some pixel value between 0~255), I first use a threshold (128) to set the given image to a binary image with pixel value 1 and 0.



Notice: F(I,j) is the original image with binarizing. First. We could compute the bond of each pixels and use bond value to check 1st LUT.

```
b = pattern[0] * 1 + pattern[1] * 2 + pattern[2] * 1 +
    pattern[3] * 2 + pattern[4] * 0 + pattern[5] * 2 +
    pattern[6] * 1 + pattern[7] * 2 + pattern[8] * 1;
```

1st pass filter:

I use “pattern” to store my pixel value in 3*3 vector and use this vector to calculate bond values. Strong connective pixels are contributed to 2(East, West, North, South), and weak connective pixels are contributed to 1. Through this method, we could quickly check the 1st LUT and see if the patterns are matched with the LUT. In order to mark the matched pixels, I use a new image-size array to mark the matched pixels. These matching positions will be marked at M. Below is part of 1st Condition LUT. (S=Shrink, T=Thin, K= Skeleton)

Table 14.3-1 Shrink, Thin and Skeletonize Conditional Mark Patterns (M=1 if hit)

Type	Bond	Patterns
S	1	0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1
S	2	0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
S	3	0 0 1 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0 1 1 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1
TK	4	0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 0

2nd pass filter:

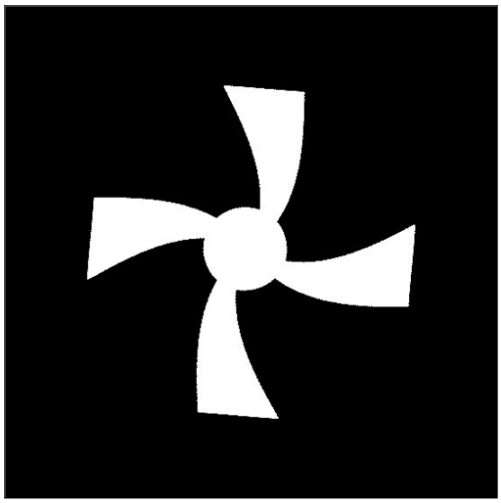
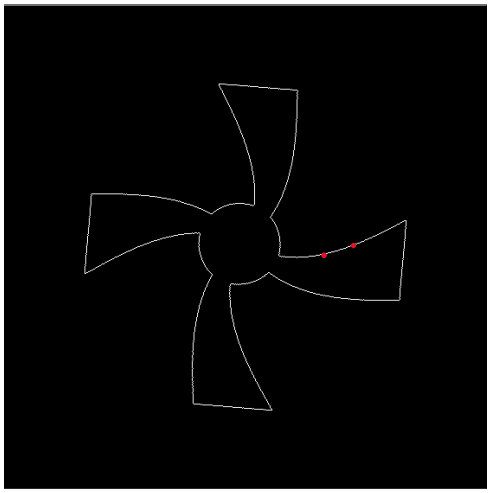
When we finish scanning all image, we could get a mapping M, the pixels in M are marked as matching pattern. We then use these matching patterns doing 2nd filter.
We will use 2nd LUT to see if these marked pixels have the matched pattern. If match, we mark hit and keep the corresponding positions pixels in the original image, if not matching, we mark miss and then erase corresponding position pixels in the original image. Usually, there are less pixel will be marked at the 2nd stage, therefore, most of pixels will be erased. Through this **recursive** process, we will finally get our shrinking (or thinning, skeletonizing image). Part of 2nd LUT (ST) is showed below:

Table 14.3-2 Shrink and Thin Unconditional Mark Patterns

Spur	0 0 M M 0 0 0 M 0 0 M 0 0 0 0 0 0 0	
Single 4-connection	0 0 0 0 0 0 0 M 0 0 M M 0 M 0 0 0 0	
L Cluster	0 0 M 0 M M M M 0 M 0 0 0 M M 0 M 0 0 M 0 M M 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 M M 0 0 M 0 0 M 0 0 M M M 0 0 M M 0 0 M M 0 0 M

I will show some example procedure in this procedure below with pictures.

For example, this is input image F(i,j) on the left, and right hand side is the image if we take input into 1st filter (1st LUT), we will get the image like this.


>>>


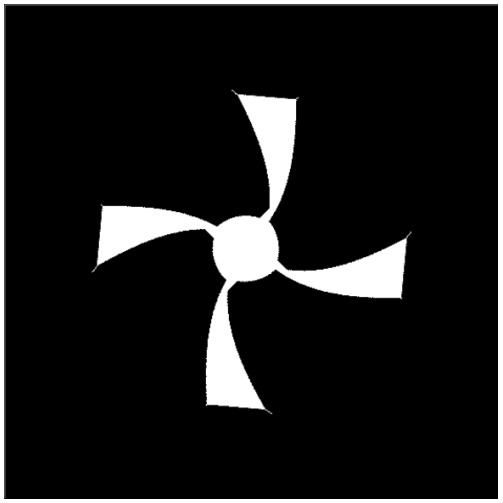
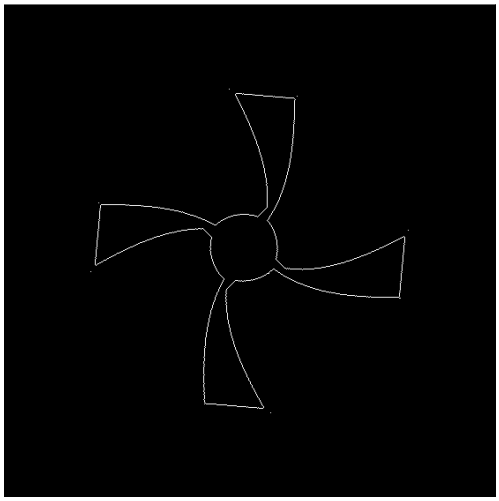
Input Image (**F**) 1st Marked **M**

Fan.raw

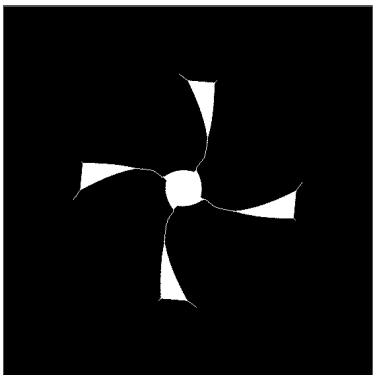
Then we take M into 2nd filter (2nd LUT), we will only mark a few pixels (only a few white dots, it's really hard to observe so I won't show it here, instead, I will mark the pixels with red dots.)

Assume we mark the red two pixels after taking M into 2nd filter, then these two red dots pixels are marked and kept. We then need to go to the input image(F) and erase the others pixel value except red pixels.

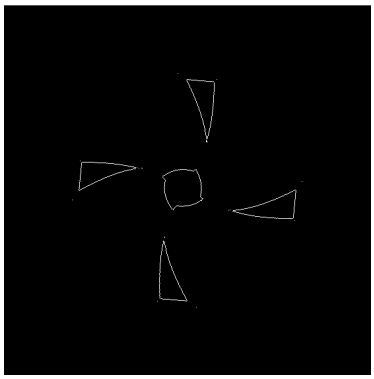
Below are the shrinking process result detail output: (Thin and skeleton are almost the same, so I only provide the process of shrinking.)


>>>


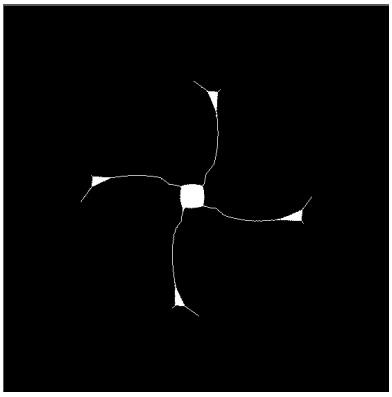
Input Image after 10 iteration shrinking Immediate result (M) after 10 iteration



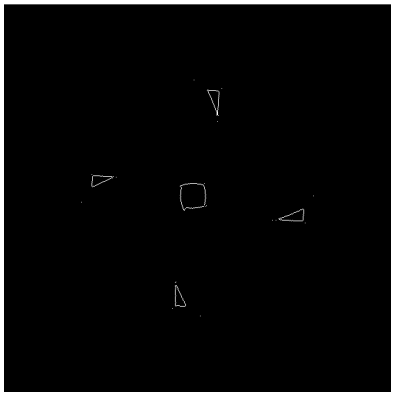
20 iterations F



20 iterations M



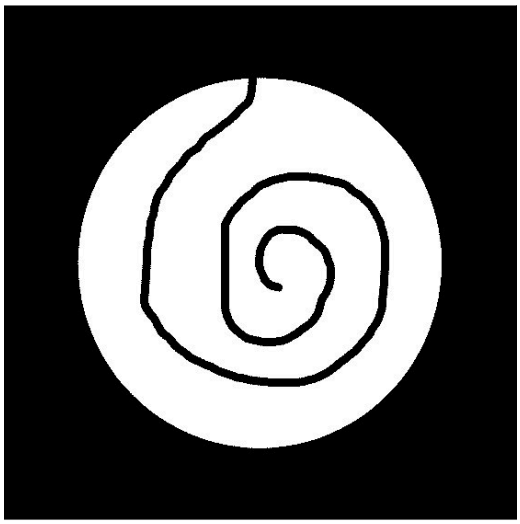
30 iterations F



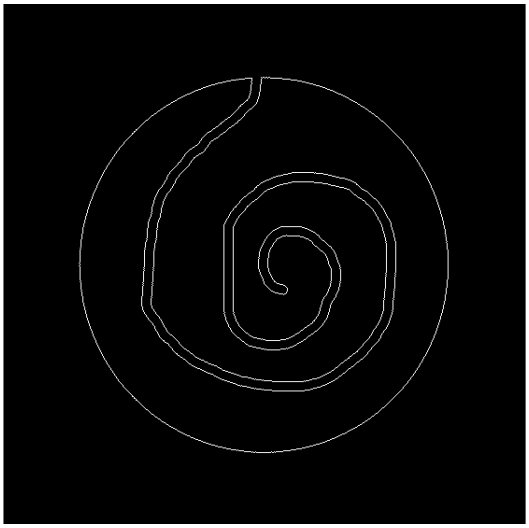
30 iterations M

We could observe the fan image has been shrunk when running more and more iterations of this procedure, and the M is also shrunk. We use recursive procedure to shrink the image until there are no more pixels to be removed.

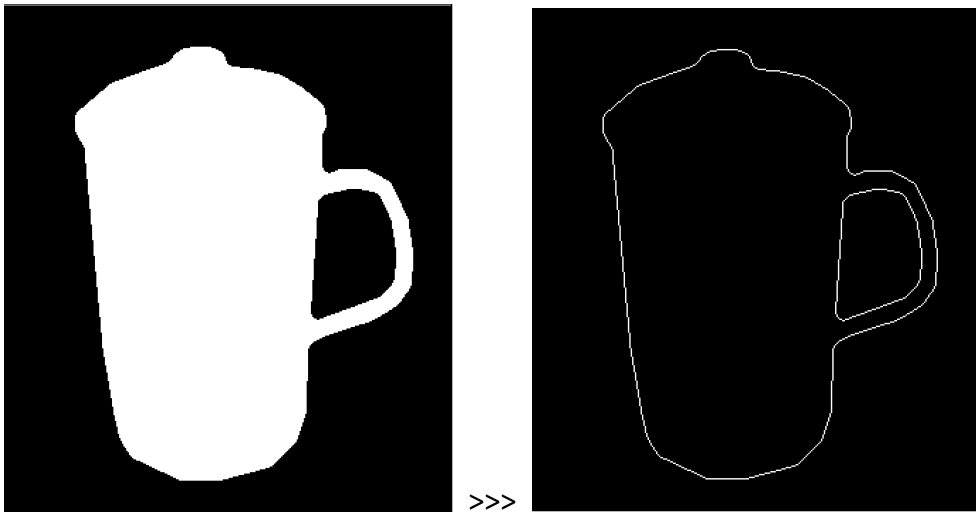
Other Immediate result are like:



>>>



Maze.raw

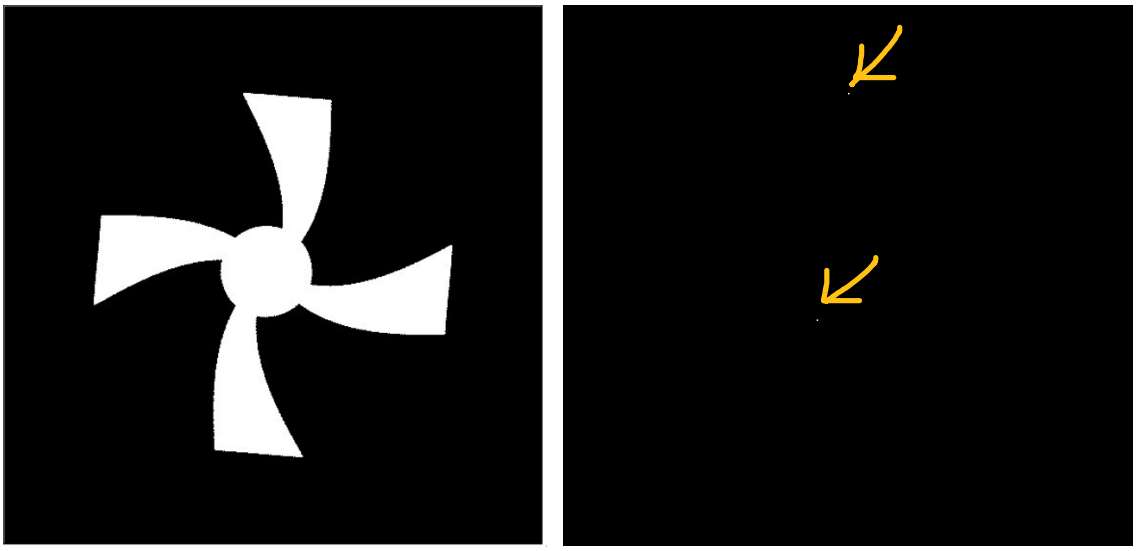


Cup.raw

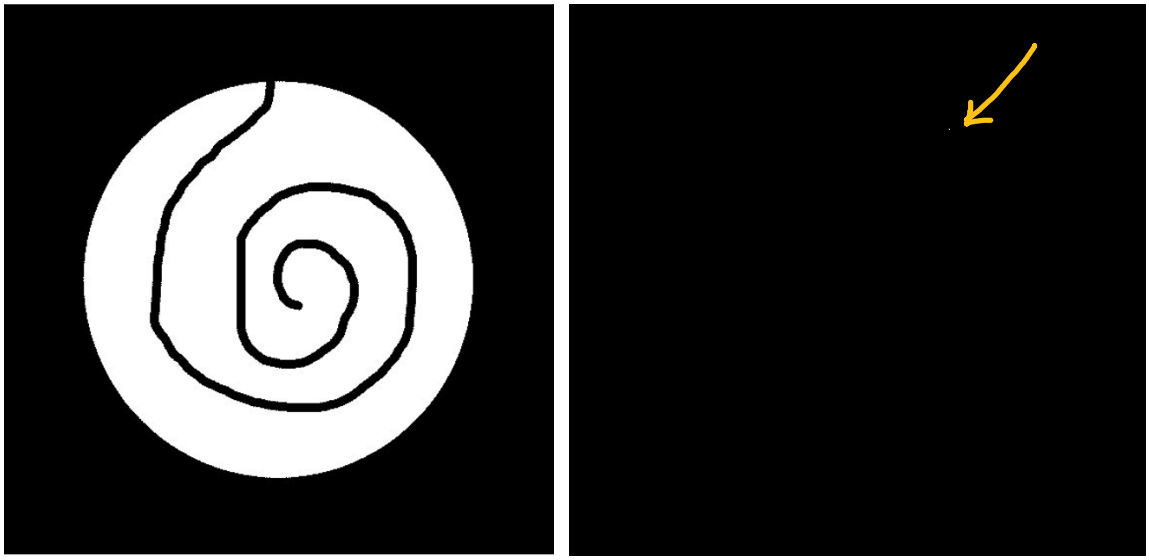
This is an example of implementing Shrinking image, Thinning and Skeletonizing are using same recursive method to implement, except using different conditional and unconditional LUT.

Result:

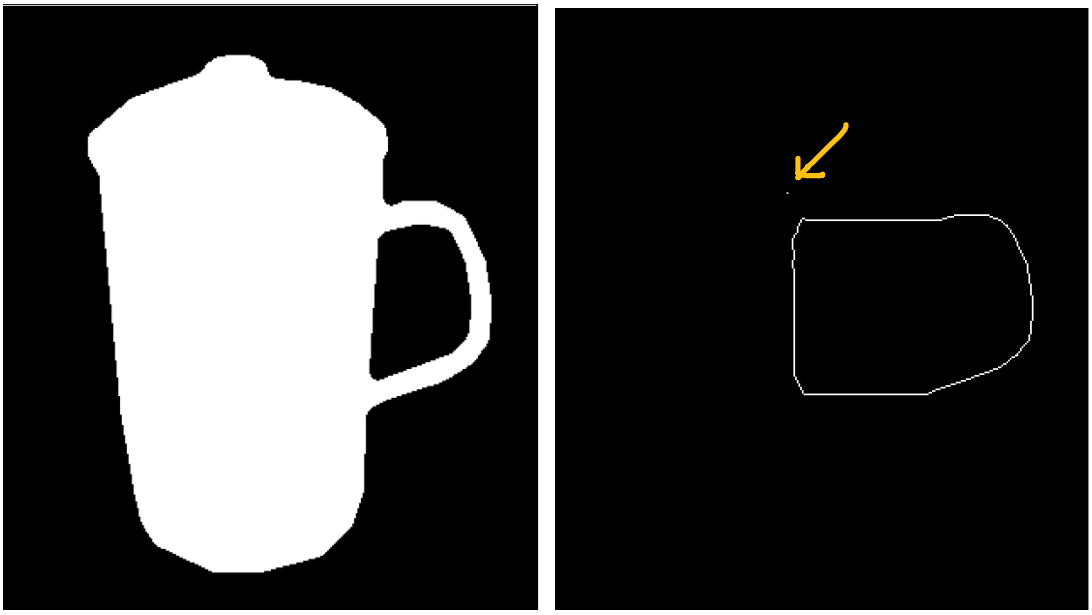
Shrinking Part: (I will use orange arrow to show the shrinking dot.)



Fan.raw

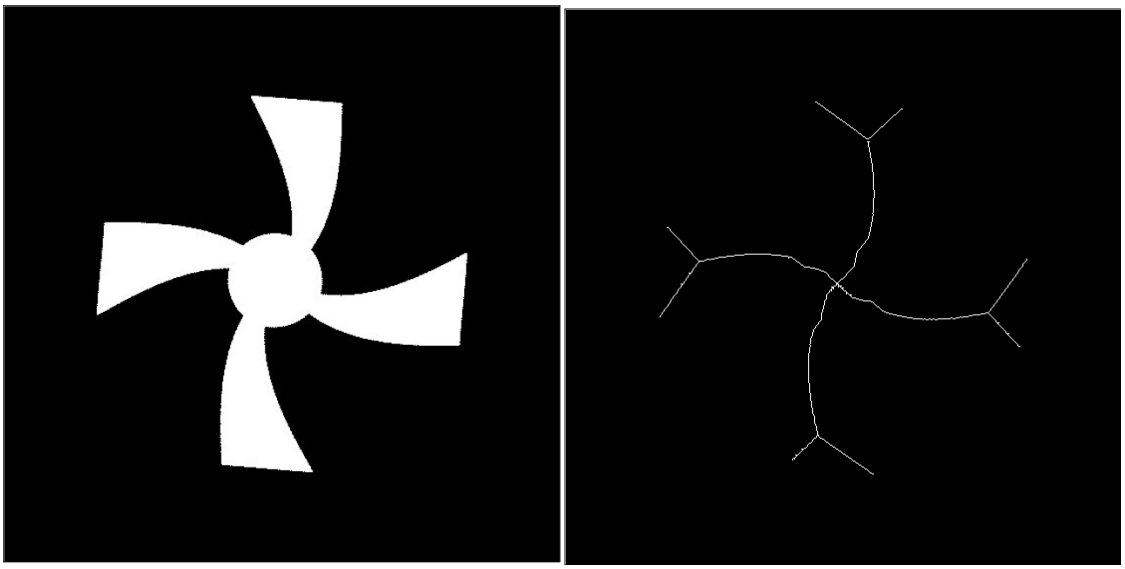


Maze.raw

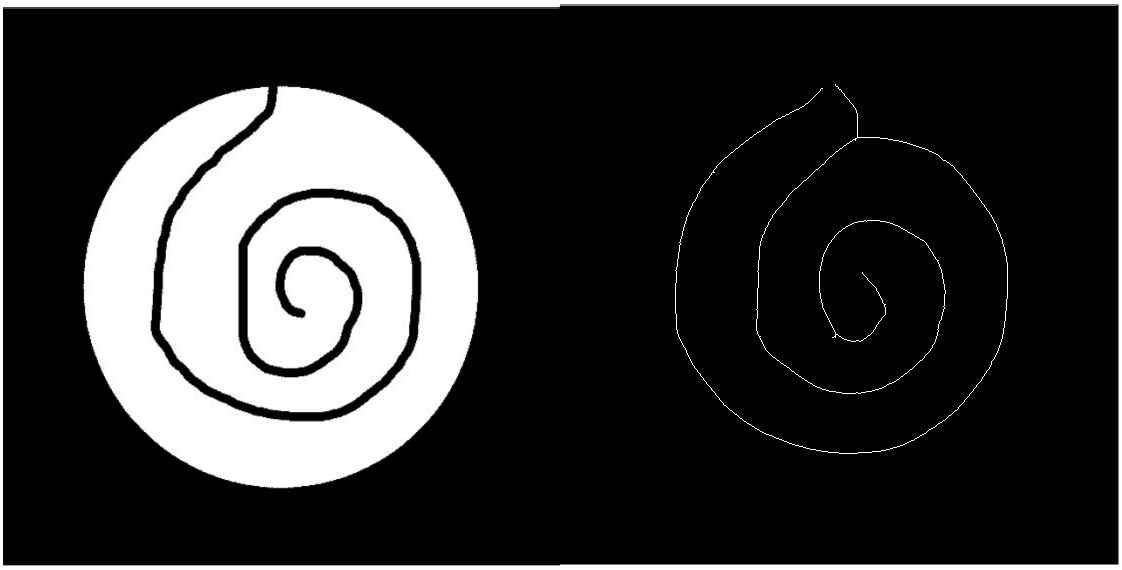


Cup.raw

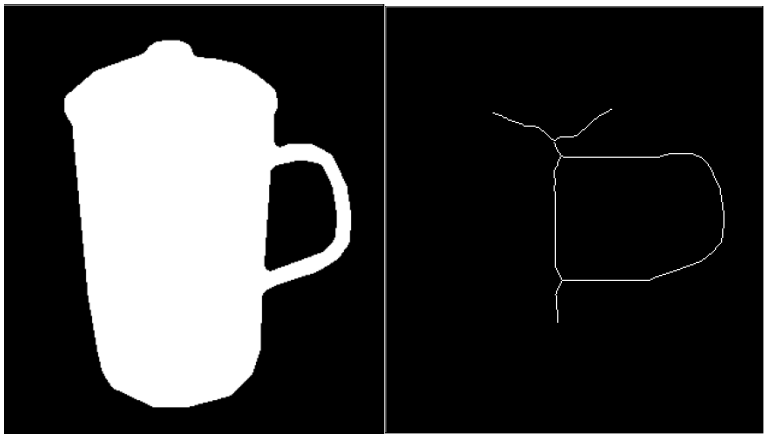
Thinning Part:



Fan.raw

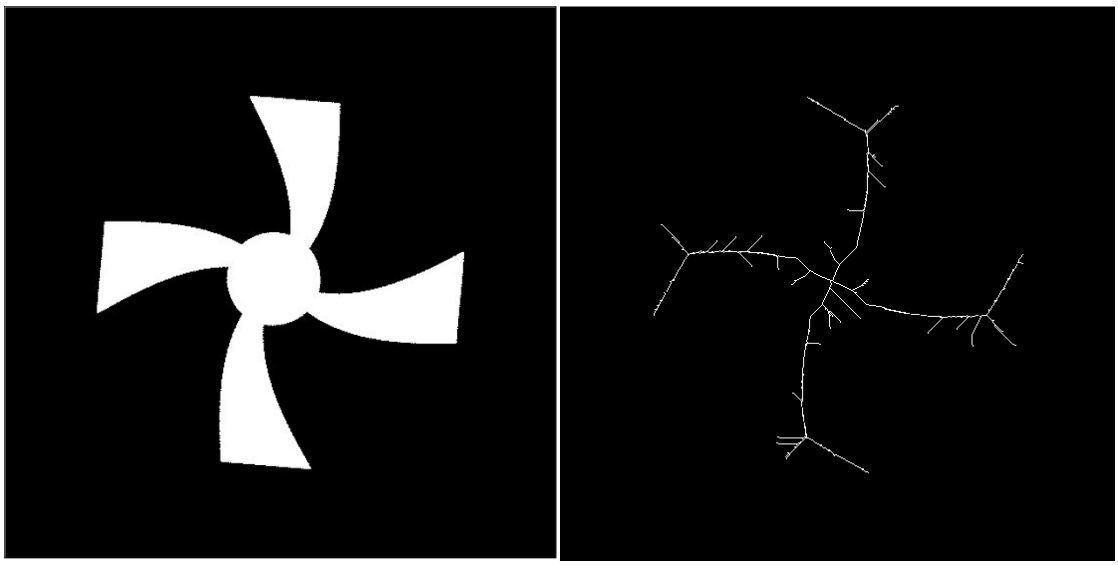


Maze.raw

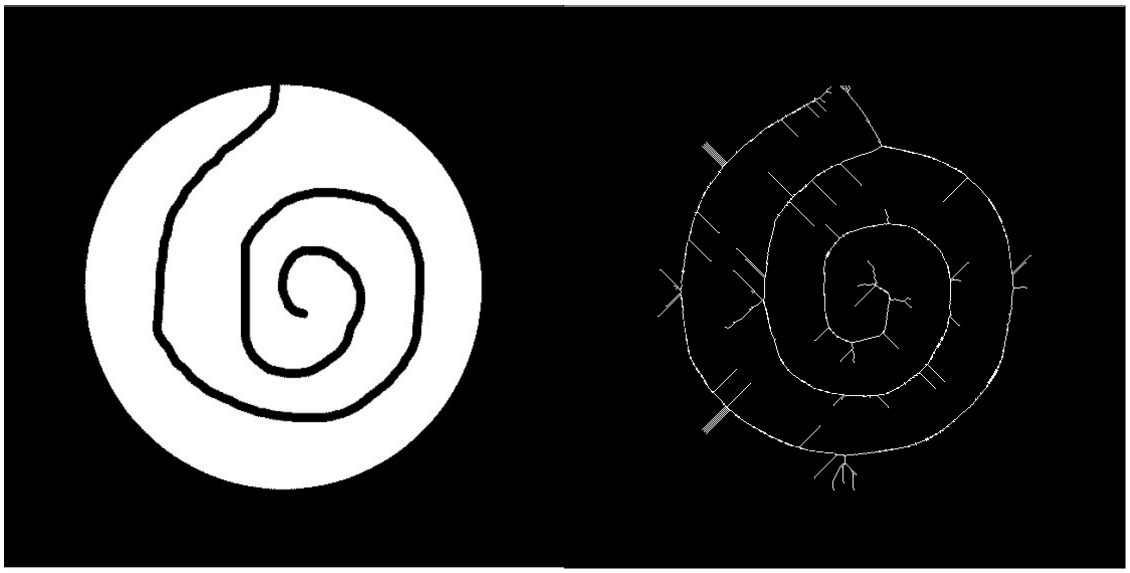


Cup.raw

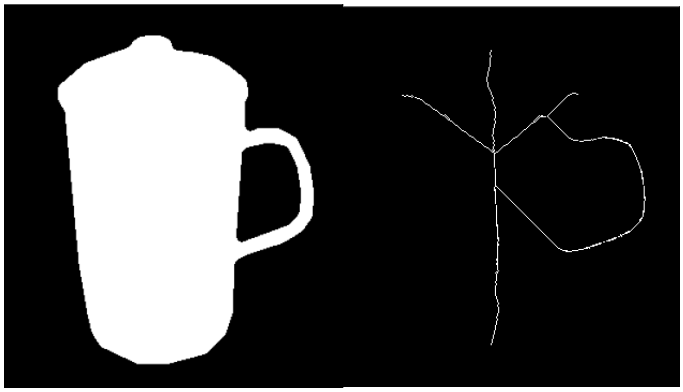
Skeletonizing Part:



Fan.raw



Maze.raw



Cup.raw

```
// fanShrink : 219 // mazeShrink : 902 // cupShrink : 106
// fanThin : 49    // mazeThin : 54    // cupThin : 90
// fanSkeleton : 35// mazeSkeleton : 45// cupSkeleton : 80
```

This is the number of iterations of different Morphological processing and different images needed to take.

2.2(b.) Counting Games - Counting Star

Note: Result will be provided when explaining approach in this part

First Algorithm – Apply Shrinking

In this problem, we need to binarize the star image first, by setting the value below 128 to 0 and greater than 128 to 1, in order to implement morphological processing. Since the stars are groups of pixels without ring and connection, we could implement **Shrinking** to shrink all stars in to one pixel. After shrinking the stars image, we could simply scan through the image and count the pixels with value (stars are the pixels with values, since background pixels are all black (value 0)).

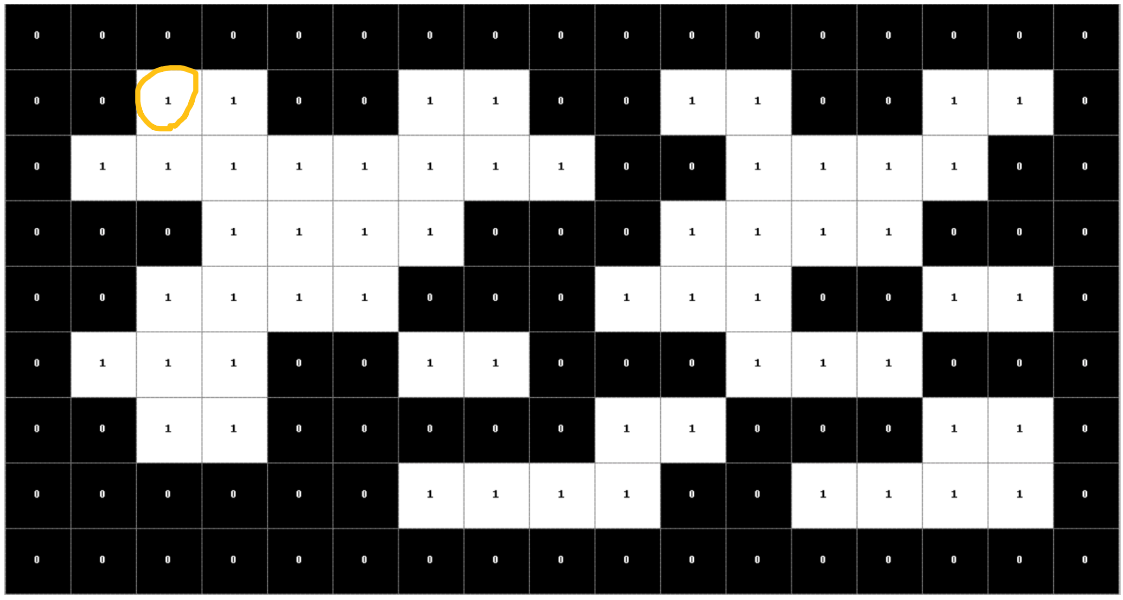
Second Algorithm – Connected Component Labeling

Notice: this method will also provide the answer of question b.(2) and b.(3)

This is a 2-pass-filter algorithm.

In this algorithm, we need to binarize the image and label every pixel first.

For example, all pixels have value one in the beginning.



Step 1: Binarize the image pixel with value 0 and 1.

1st – pass filter

Step 2: We need to use a L-Mask to assign new label to each pixel.

There are two different situations when assigning labels. Since, the beginning pixel value is 1, we will start assigning label with value two.

1st situation:

0	0	0
0	1->2	

If L-Mask are all backgrounds, we will assign a new label (here is 2) to the center pixel. One thing need to be noticed that after a new label is assigned, the next same situation we will assign.

2nd situation:

3	5	5
4	1->3	

If there are values in L-Mask, we will choose the minimum value (here is 3 for example), then we need to assign this minimum value to the center pixel.

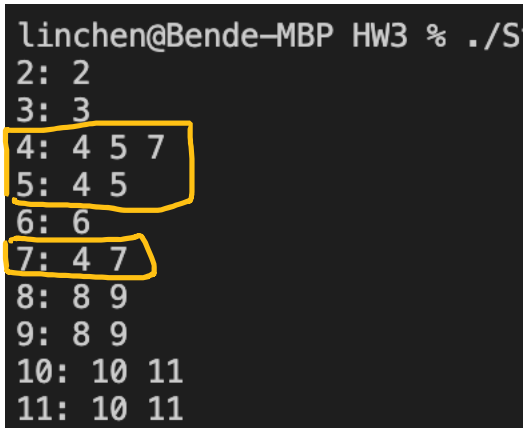
Step 3: After step 2, we will get a group of pixels with different values. For example, there are more than one label in the same star.

For example, one of the stars

0			4
0	4	4	0
5	5	5	0
0	7	0	0

Then we need to merge the pixel labels in the same star with the minimum label. (here is 8 for example). In order to merge the label, I use a map data structure in C++ with **key are labels** and **values are the sets of labels** to merge the label.

The map will look like this:



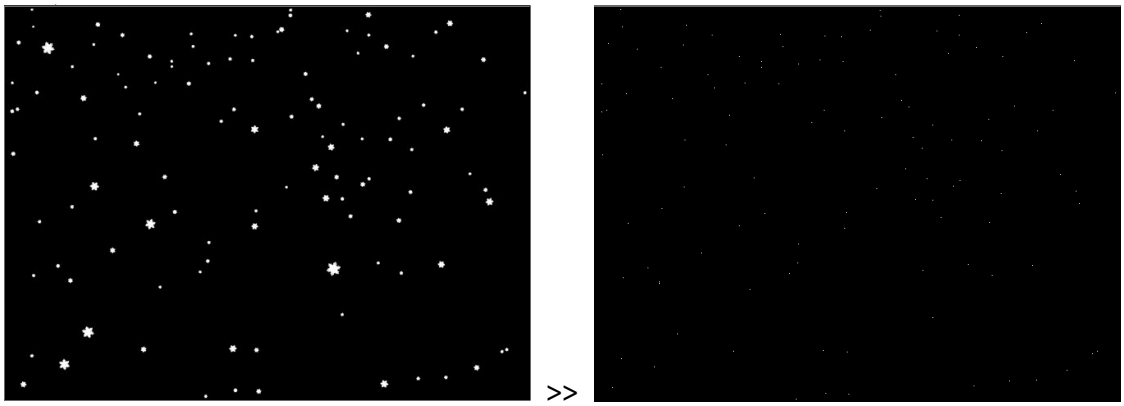
This map means the which labels are in the same star (tale above image for example, 4,5,7 are the labels within the same star)

Step 4: (Merge Labels) With the map I created in step 3, I could start merging the labels within the same star. Merging step is easy once created the map, just scan the whole image again and set the pixels to the minimum connected label. Then we will have the same labels within each star.

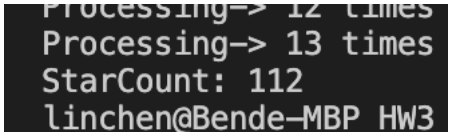
Step 5: (Size and Frequency) After step 4, I created an array to store size of each different labeled star. The indices of array are the labels of the star, and the values are their size. Then I used a map data type to count the frequency of size and print the result.

Result:

1st Algorithm (Shrink):

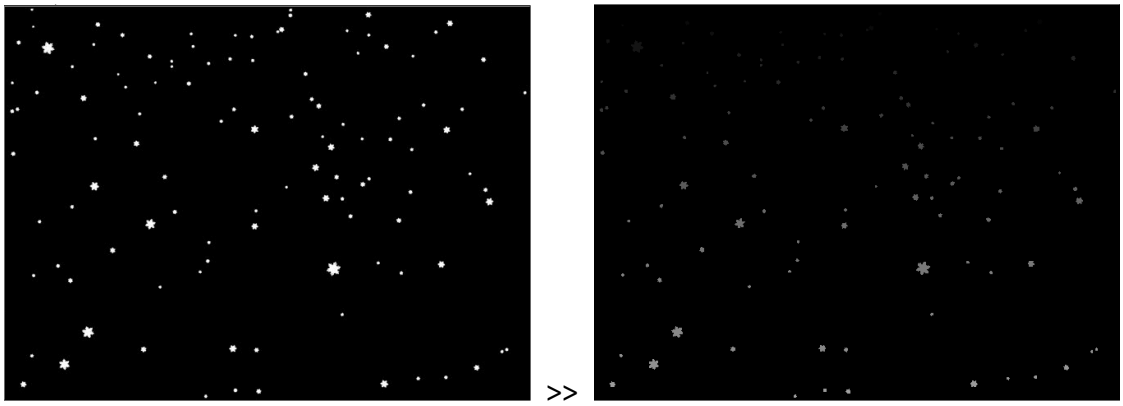


Original Image >> Image after shrinking



Count the white dots in the image above, we could know the number of stars is **112**

2nd Algorithm (Connected Component Labeling):



Original Image >> Labeled Stars Image

Each star in labeled Stars image has different pixel value. Since I scan the image from up to down, therefore, we could observe that the stars are getting brighter and brighter.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

linchen@Bende-MBP HW3 % g++ -std=c++11 Star.cpp -o Star
linchen@Bende-MBP HW3 % ./Star stars.raw starsCount.raw
starCount: 112
Different sizes: 41
Size | Frequency
  1  :  2
  5  :  2
  6  :  1
  7  :  2
  8  :  7
  9  :  9
  10  :  8
  11  :  6
  12  :  8
  13  :  10
  14  :  2
  15  :  7
  16  :  3
  17  :  6
  18  :  1
  19  :  1
  20  :  6
  21  :  1
  22  :  3
  23  :  1
  24  :  2
  25  :  2
  27  :  1
  28  :  1
  30  :  2
  32  :  1
  33  :  1
  34  :  1
  38  :  2
  40  :  1
  41  :  1
  43  :  1
  44  :  2
  48  :  1
  49  :  2
  54  :  1
  66  :  1
  84  :  1
  96  :  1
  114  :  1
  129  :  1
  155  :  1
linchen@Bende-MBP HW3 %
```

Size and Frequency of stars

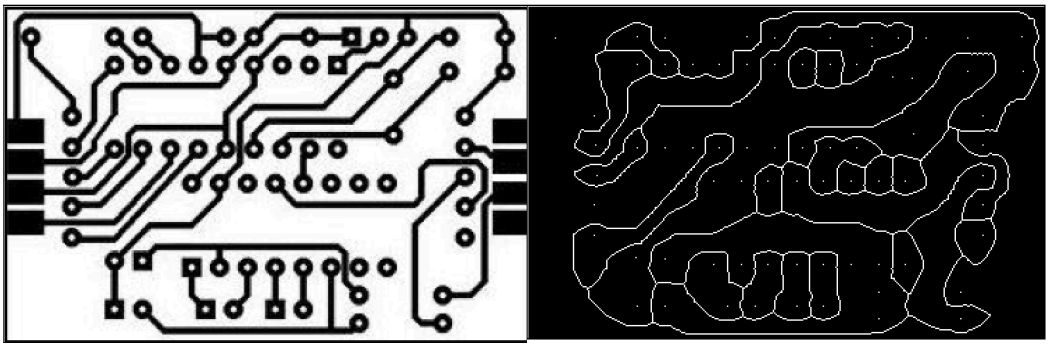
2.2(c.) PCB analysis

In order to solve this problem, we need to first understand that what **Shrinking** in different objects. When implementing shrinking, objects without holes shrink to a **point**, and objects with holes shrink to a **connected ring**. Know these, then we can start solving this problem.

Step 1: Binarize the input image (with threshold pixel value 60), the reason of choosing 60 instead of 128 I will explain later in the discussion part.

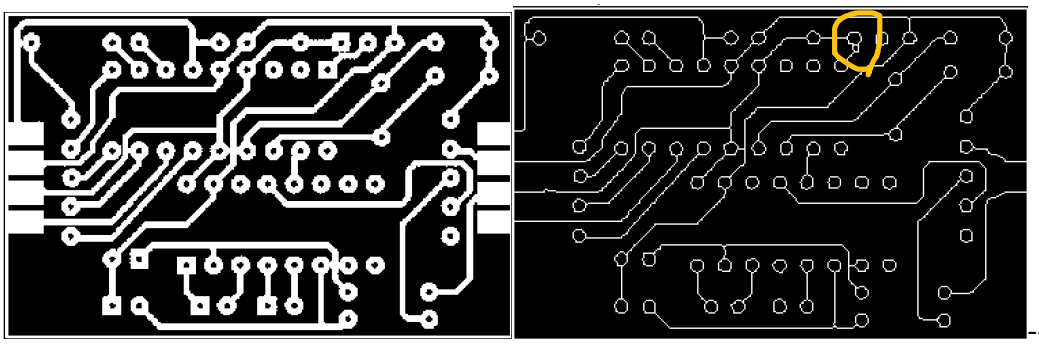
Step 2: Shrink this binarized image and get: (1st shrink)

Notice: the dots here represent the holes in the image



---A

Step 3: Reverse the image from black to white and white to black, then shrink this image, then we will get: (2nd Shrink) By shrinking this "reverse image", we could get keep the holes and pathways.

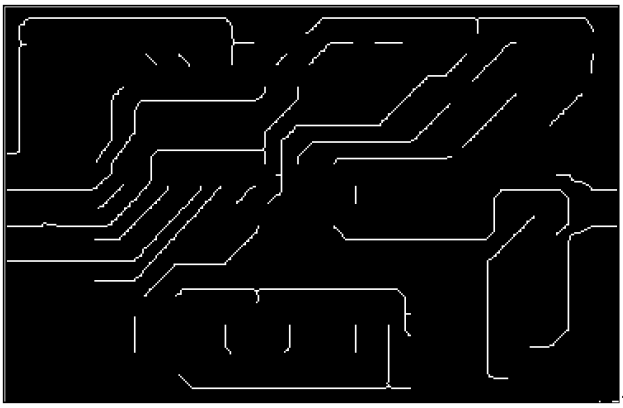


---B

Step 4: From the 2 images above, we could actually start counting the holes in the image in the first image. In order to counting the line, we should do one more shrink. I use a 3*3 mask to count the hole, that is, when the center pixel has pixel value 1 and others don't, then count 1 circle, and mark the position of the circle. Since we have marked the positions of where the centers of circles are, we could erase the circle in picture B by setting the neighborhood pixels value to 0. (I use radius 6, therefore, if we include the center point the mask size will be 13*13, that is, set the neighborhood pixel of the

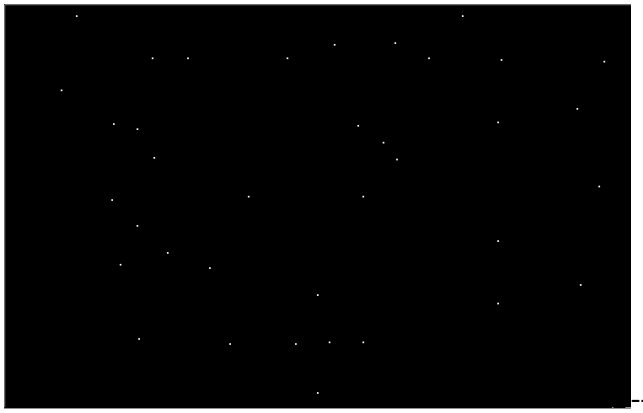
center of circles to zero in a 13*13 mask)

After this procedure, we could get the image like this:



---C

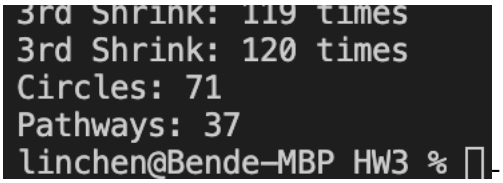
Step 5: Do one more shrink on C (3rd shrink), since all the lines will be shrunk to dots. We could easily calculate the amount of the dots by scanning the image and find the pixel with value 1.



---D

Notice: The remaining dots in this image are pathways.

After using scanning the image in image A and image D, we could finally get the total number of circles and pathways.



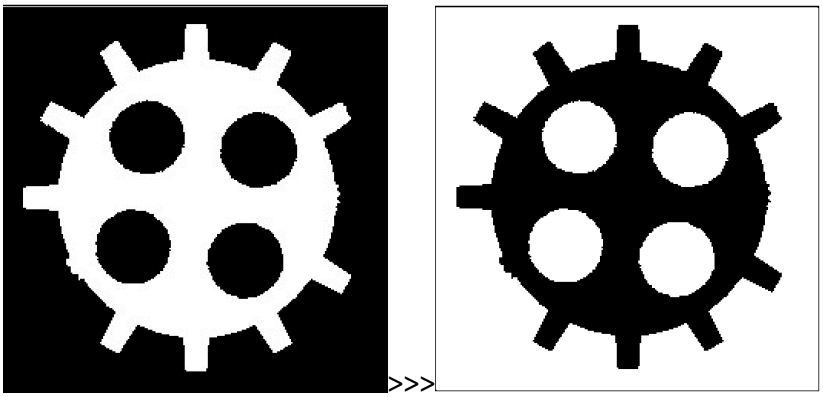
final answer

2.2(d.) Defect Detection

In this problem, we need to detect the two missing teeth around a normal gear.

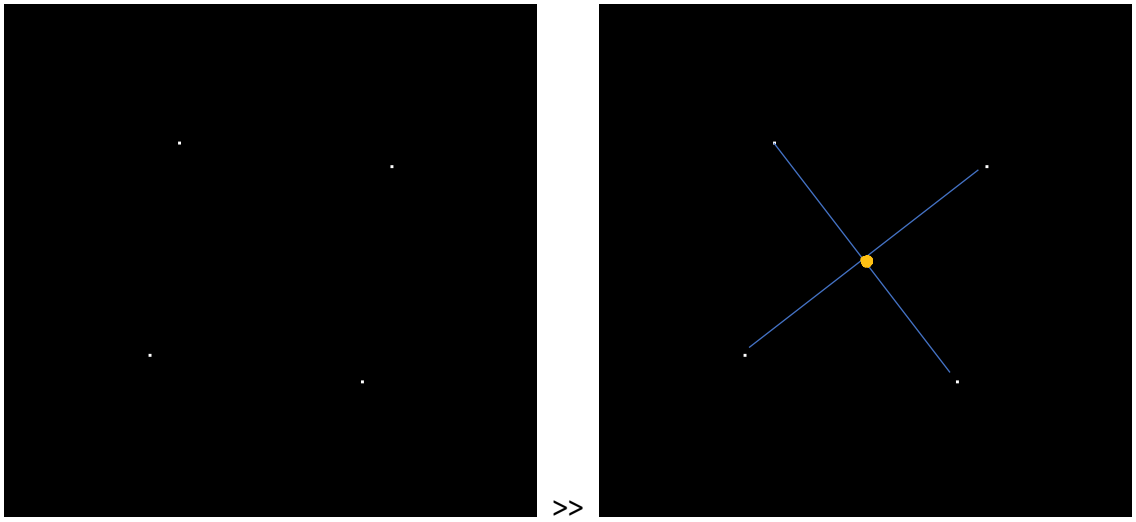
In order to detect the teeth, we need to first calculate the center position of the gear and the radius of the gear.

Step 1: Binarize the GearTooth image, and then reverse it black to white, white to black.



Then we implement shrinking to the reverse image.
Since there are four circles in the gear, if we implement shrink on it, we will get 4 points, we use these 4 points to find the center position of the gear, which is approximately (124,124).

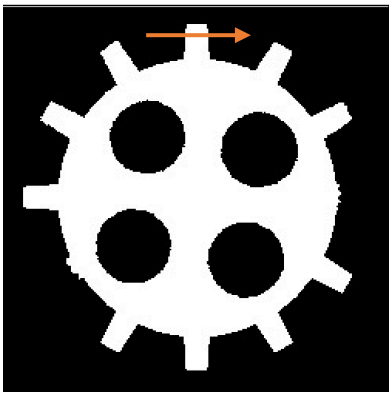
Step 2: Find the center and the radius of the gear.



Once we find the center position, radius could be easily to calculate. I use this position (124,124) in the image with a for-Loop, fix the x-position and keep changing the y-position, in the end, I found that the radius is 113 pixels. When knowing the center and the radius, we could start detecting our teeth.

Step 3: Since the given description of the problem says that the teeth are uniformly distributed, every tooth should be 30 degrees away from each other. I use cos and sin with radius to extend the position start from center (125,125) to get the corresponding

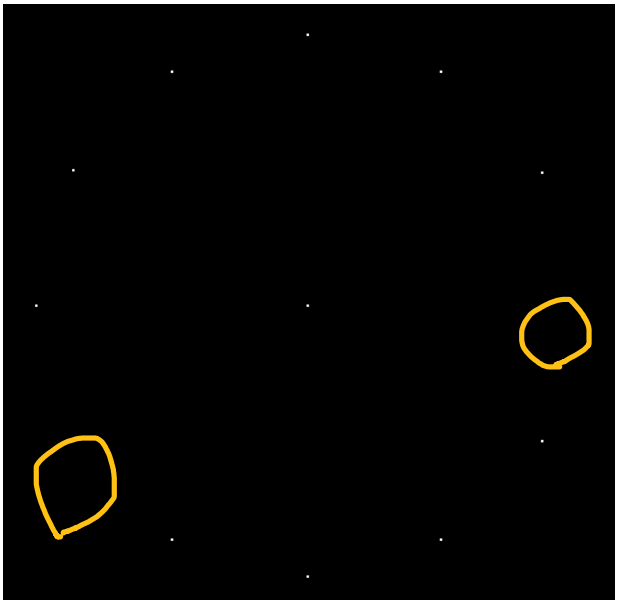
position of each teeth.



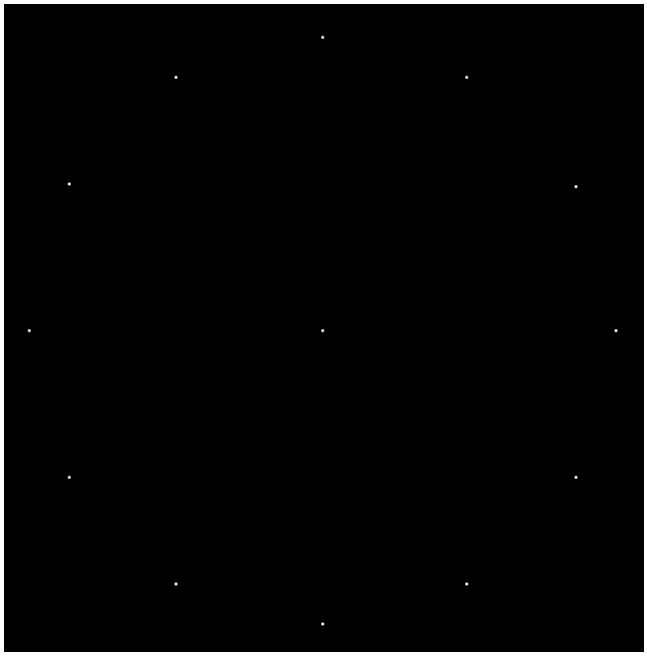
Clock-wisely detect the tooth with below function

```
for(double i = 0; i < 360; i+=30) {
    X_dir = radius * cos(i * PI / 180);
    Y_dir = radius * sin(i * PI / 180);
    if(ImagedataB[125+int(X_dir)][125+int(Y_dir)][0]==1) {
        TeethCnt++;
        ImagedataDetect[125+int(X_dir)][125+int(Y_dir)][0] = 1;
    }
    else {
        std::cout << "Miss Teeth position: " << "row: " << 125+int(X_dir) << " ,col: " << 125+int(Y_dir) << std::endl;
    }
    ImagedataCircle[125+int(X_dir)][125+int(Y_dir)][0] = 255;
}
```

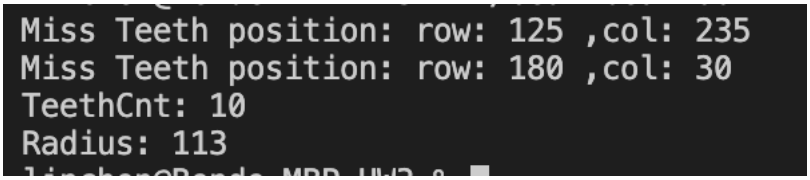
Result:



Yellow circles (without white dot) are the missing teeth.



If we add the missing teeth, the final completed gear teeth will be like this.



Miss Position and detection result.

2.3 Discussion

2.3 (b.) Counting Games

1. the total number of the star is: **112**

2. There are **41** different sizes of star. The frequency of the star has been provided above in the approach and result part.

3. The other method is Connected Component Labeling, the detail and the result are also provided above in the approach and result part.

2.3 (c.) PCB analysis

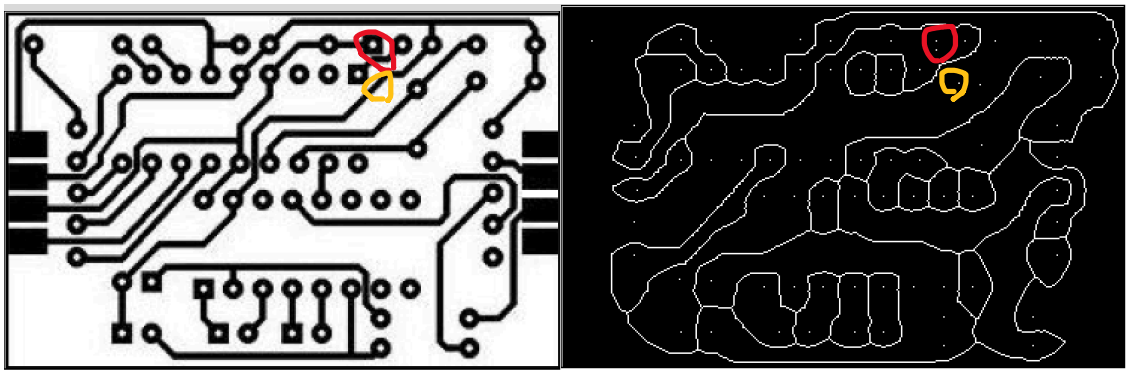
1. There are **71** holes in the given PCB.

2. There are **37** pathways in the given PCB

Discussion part:

The algorithm what I used and the details of every procedure I have already explained above in approach and result part. I will provide some of my observation when solving this problem.

In the beginning, I counted the number of holes by hand which was 71, but when I used program to run, the number of holes was 73. Then I checked the procedure I have processed. I found that there were some parts needed to be carefully noticed.



Since the holes are really closed to the pathways, when implementing binarizing, I might accidently connect the boundary of the circle to the pathway, which will create additional dots when implementing shrinking.

```
Circles: 73
Pathways: 34
```

This is the original result when I use 128 as a threshold to binarize the image. Additional circles will also reduce the number of pathways. Then I go back to original image and see nearby pixels value of the position I just marked above and decided to set the threshold to 60 when binarize the image. Then I could get the final result:

```
3rd Shrink: 119 times
3rd Shrink: 120 times
Circles: 71
Pathways: 37
linchen@Bende-MBP HW3 %
```

2.3 (d.) Defect Detection

Details of the solution have already provided in the approach and result part!

```
Miss Teeth position: row: 125 ,col: 235
Miss Teeth position: row: 180 ,col: 30
TeethCnt: 10
Radius: 113
```

<< answer

This result using center (125,125) and radius = 110 (I lower a little bit the radius to make sure that the detecting position will be located in the teeth.)