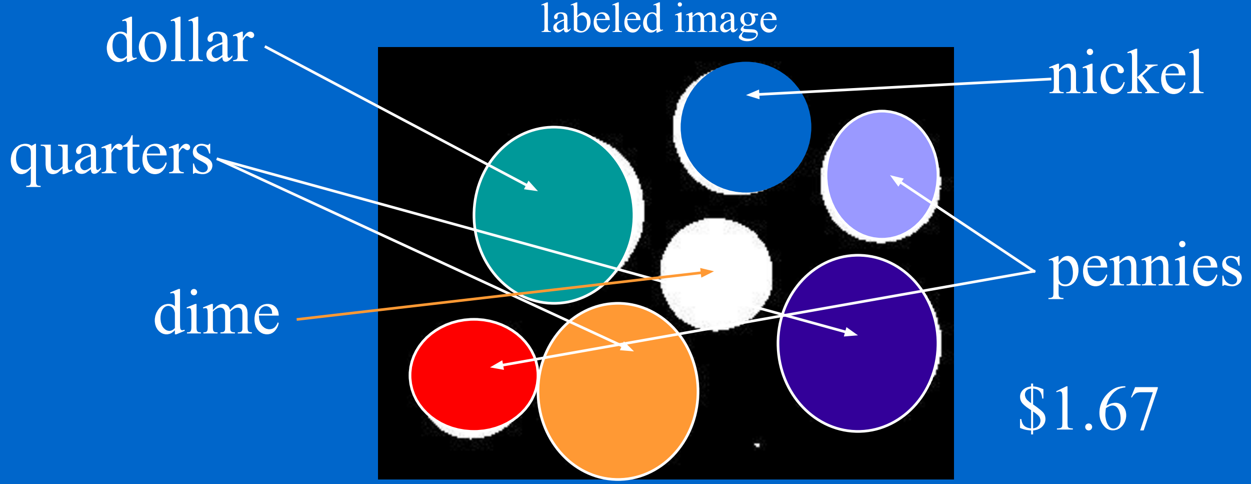
Programming Assignment 2

CS 308

Assignments 2&3: Build a Simple System to Recognize Coins

original image

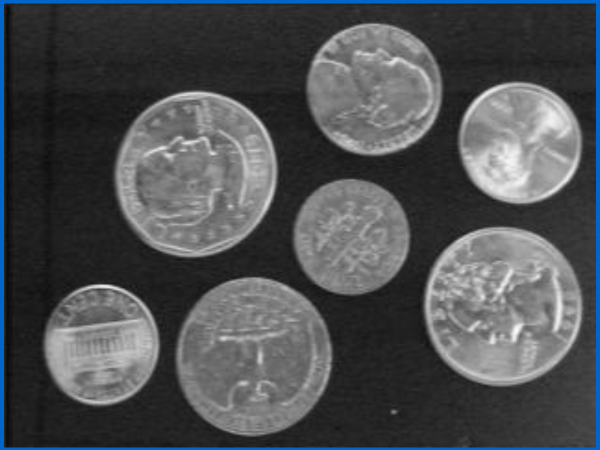
labeled image



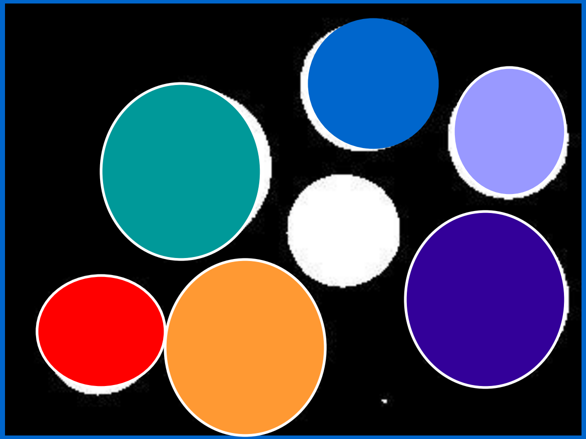
$1.67

Assign 2: Label and Count Regions

original image



labeled image

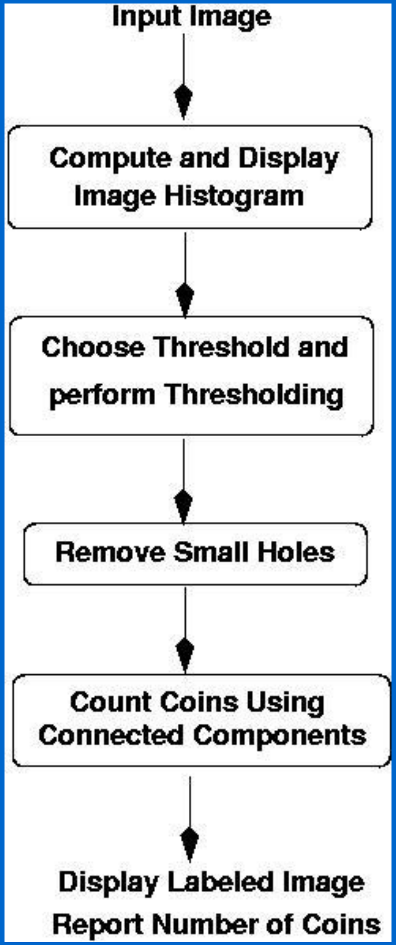


→ 7 regions

Project Objectives

- Improve your skills with manipulating stacks and queues.
- Improve your understanding of recursion.
- Illustrate how to convert a recursive algorithm to an iterative one.
- Learn more about image processing.
- Learn to document and describe your programs

Flowchart for labeling and counting regions


Input Image
Compute and Display Image Histogram
Choose Threshold and perform Thresholding
Remove Small Holes
Count Coins Using Connected Components
Display Labeled Image
Report Number of Coins

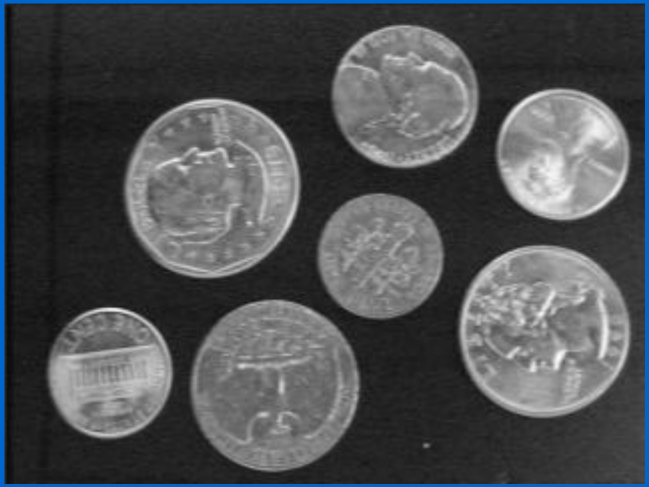
(1) add a new option to your menu called "Count/Label Regions"

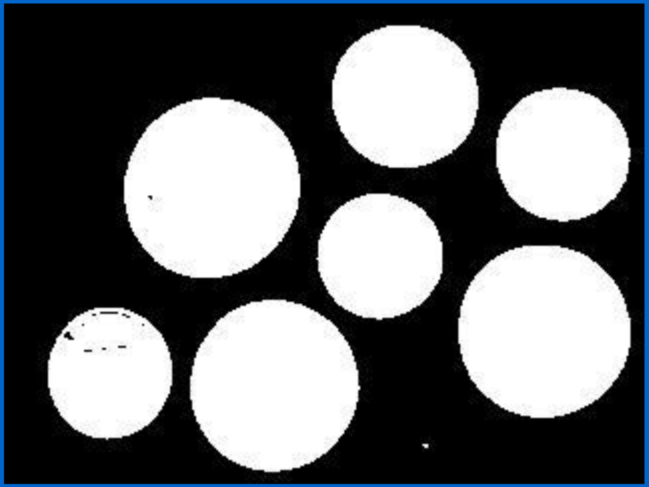
(2) the steps given in the diagram should be executed when the user selects this option.

(you can also have these steps as separate menu options)

Thresholding

- Generates a binary (black/white) image of the input.
- Separates the regions corresponding to the coins from the background.
- Segmentation is useful for performing coin recognition:
  - collect all the pixels belonging to the same region
  - extract “features” useful for coin recognition





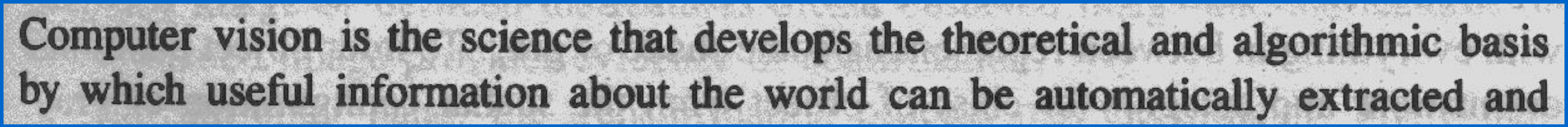
threshold(image, thresh)

- Implement it as a client function (only for grayscale images).
- Each pixel in the input image is compared against a threshold.
- Values greater than the threshold are set to 255, while values less than the threshold are set to 0.

$$O(i,j) = \begin{cases} 255 & if \quad I(i,j) > T \\ 0 & if \quad I(i,j) \leq T \end{cases}$$

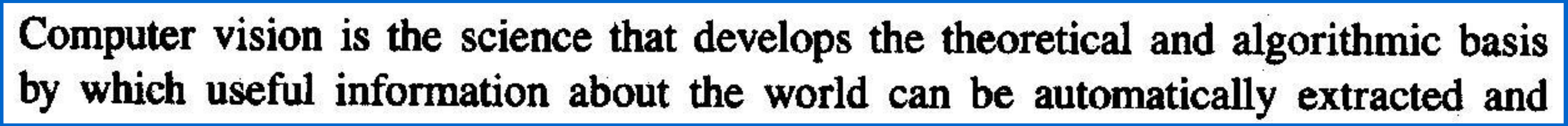
Other Examples: Character segmentation

original



Computer vision is the science that develops the theoretical and algorithmic basis by which useful information about the world can be automatically extracted and

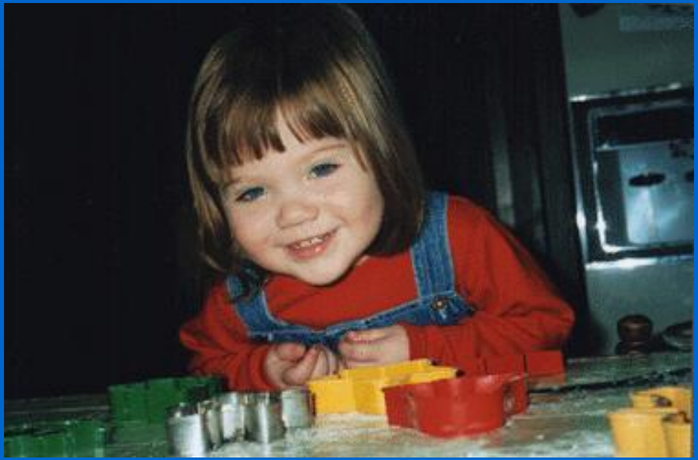
thresholoded



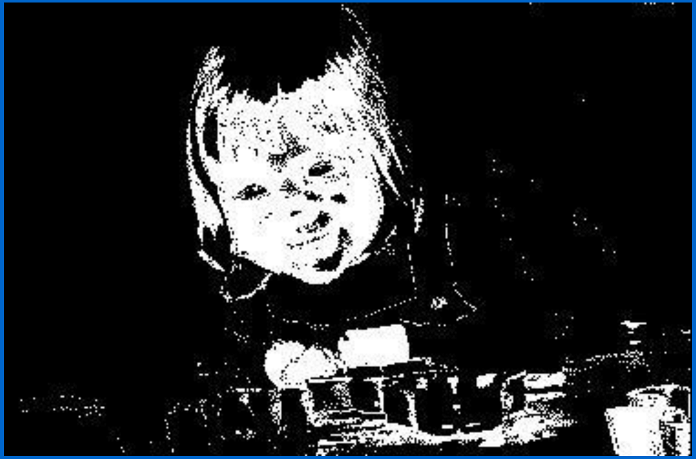
Computer vision is the science that develops the theoretical and algorithmic basis by which useful information about the world can be automatically extracted and

Other Examples: Face segmentation

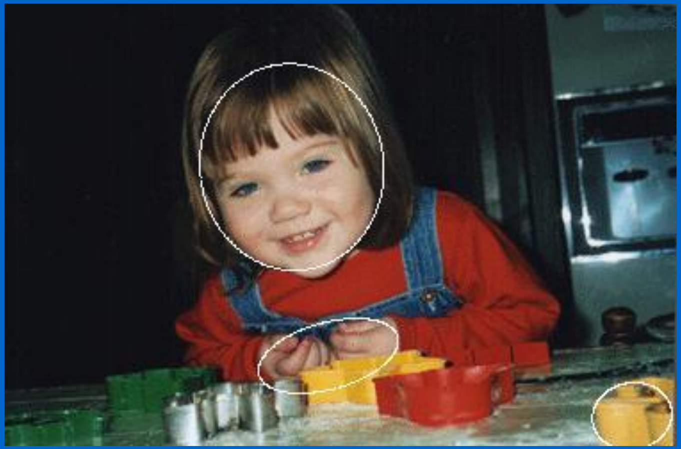
original



thresholded

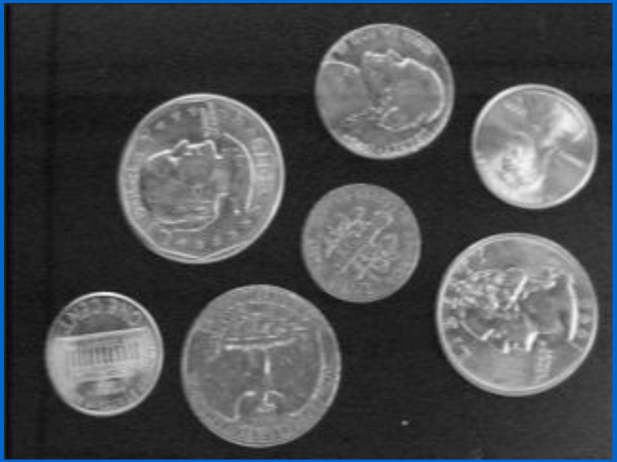


candidate face regions

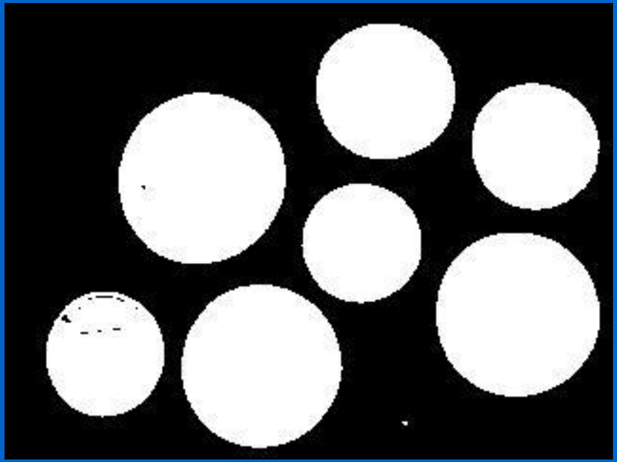


How to choose the threshold?

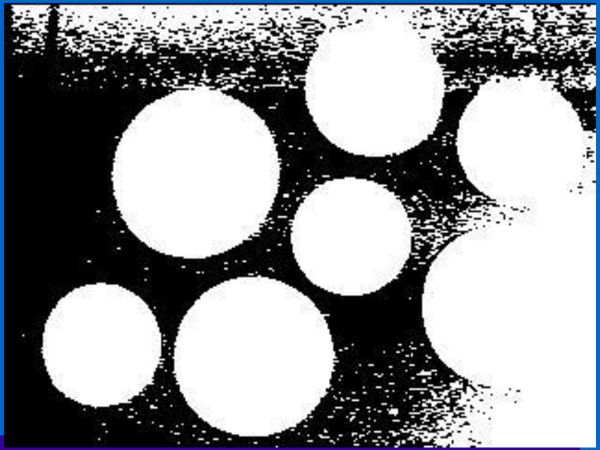
original



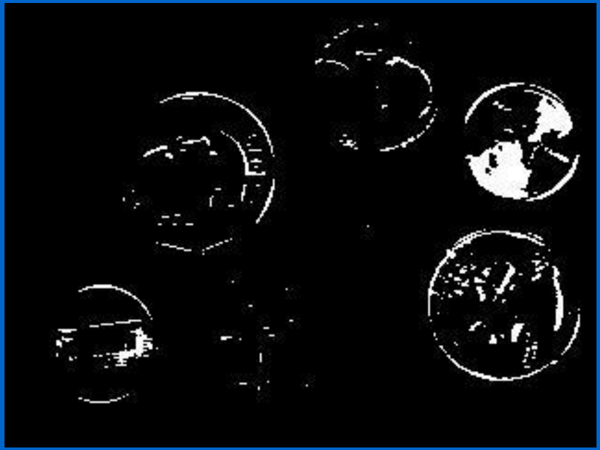
good threshold



low threshold



high threshold



displayHistogram(image)

- Implement it as a client function.

- The histogram is a bar graph of the pixel value frequencies (i.e., the number of times each value occurs in the image)

0	0	1	0	2	0
1	0	7	7	7	0
0	7	0	0	7	0
1	0	0	7	2	0
0	0	7	1	0	1
1	0	7	7	7	0

frequencies

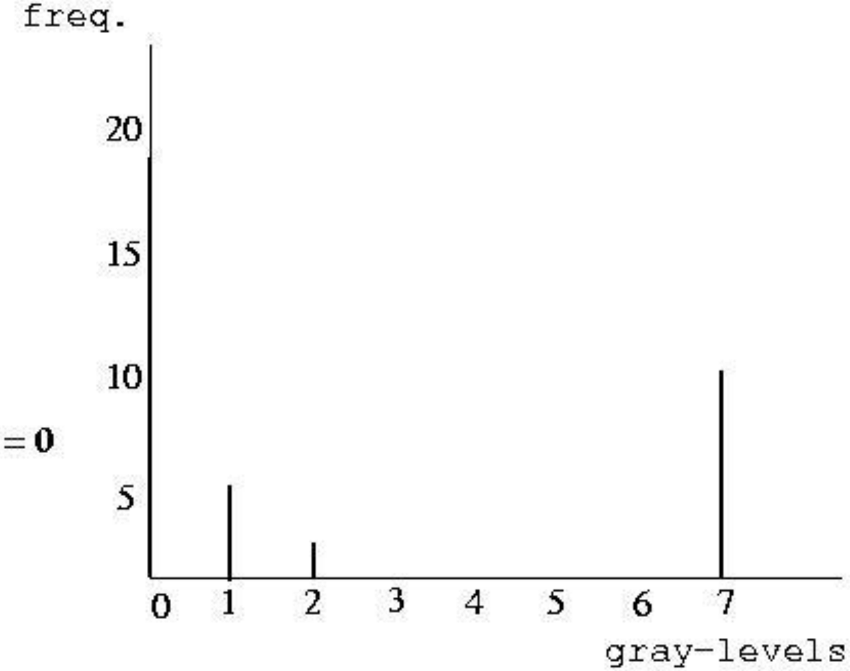
f(0)= 18

f(1)= 6

f(2)= 2

f(3)=f(4)=f(5)=f(6) = 0

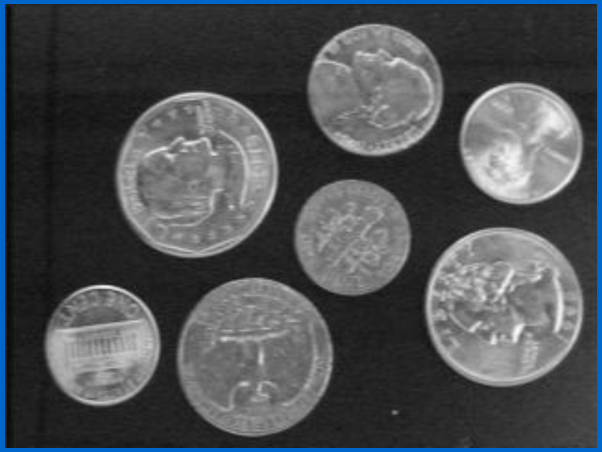
f(7) = 10

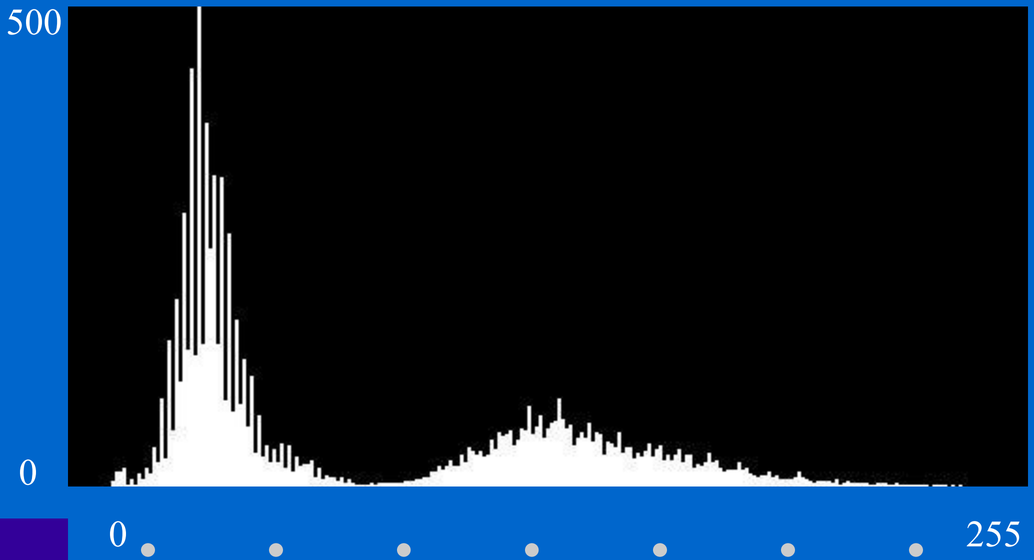


displayHistogram(image) -- cont'd

- Use an array of counters to store the pixel frequencies.
- Display the histogram as an intensity image.
  - Draw a bar for every counter.
  - Normalize counter values:

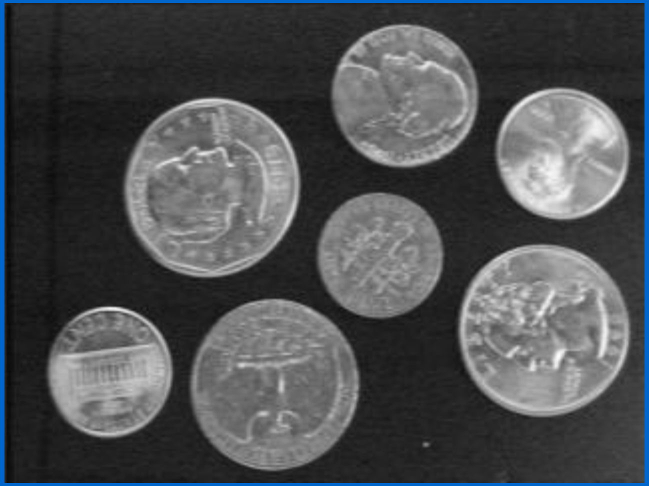
$$\bar{c} = \frac{c}{\max\_c} \times 500$$

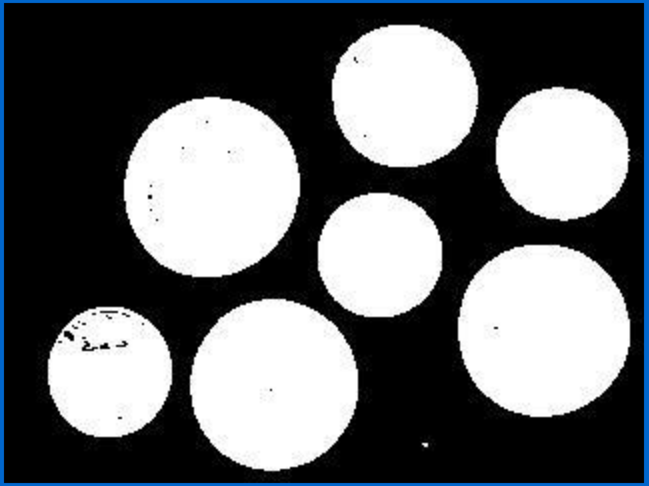




Improving the results of thresholding

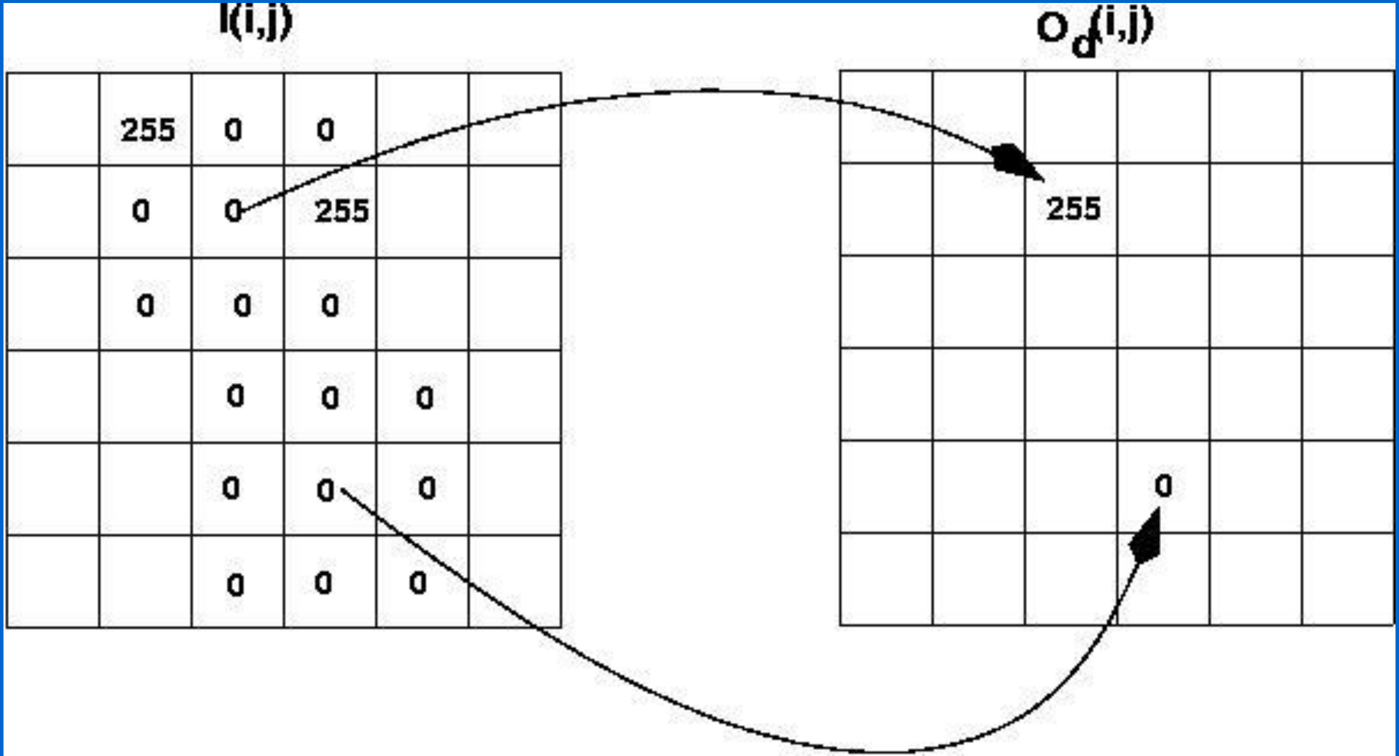
- In most cases, further processing is required to improve the results of thresholding.
- For example, some of the regions in the thresholded image might contain holes.





dilate(image) -- client function

$$O_d(i,j) = \begin{cases} 255 & \text{if} \quad \text{at least one neighbor is 255} \\ I(i,j) & \text{if} \quad \text{all 8 neighbors are 0} \end{cases}$$



I(i,j)

	255	0	0		
	0	0	255		
	0	0	0		
		0	0	0	
		0	0	0	
		0	0	0	

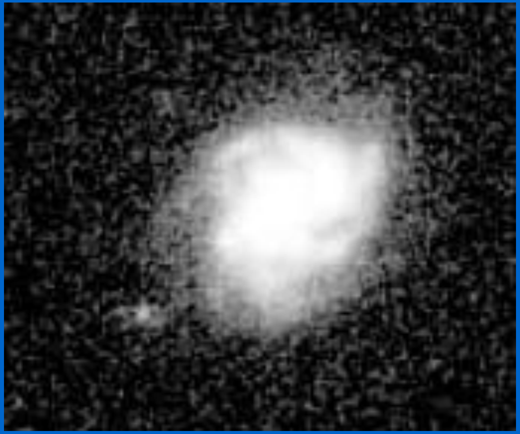
$O_d(i,j)$

		255			
			0		

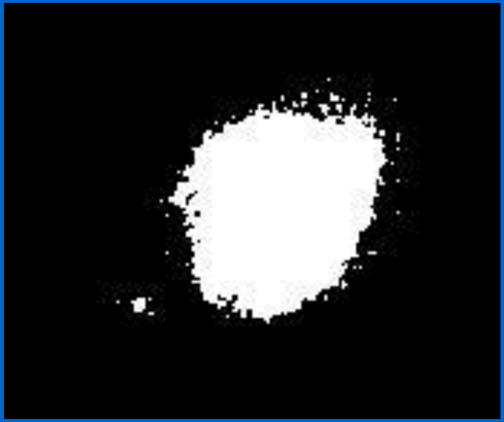
dilate(cont'd)

- Dilation "expands" the regions (i.e.,adds a layer of boundary pixels)

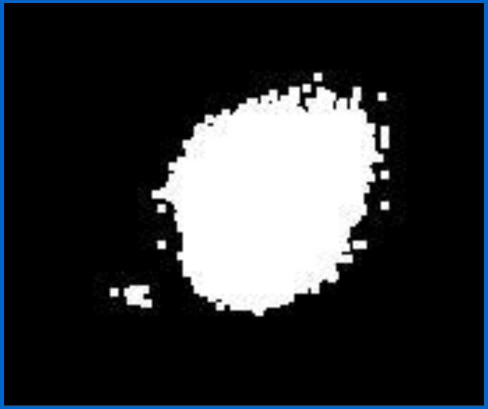
original



thresholded

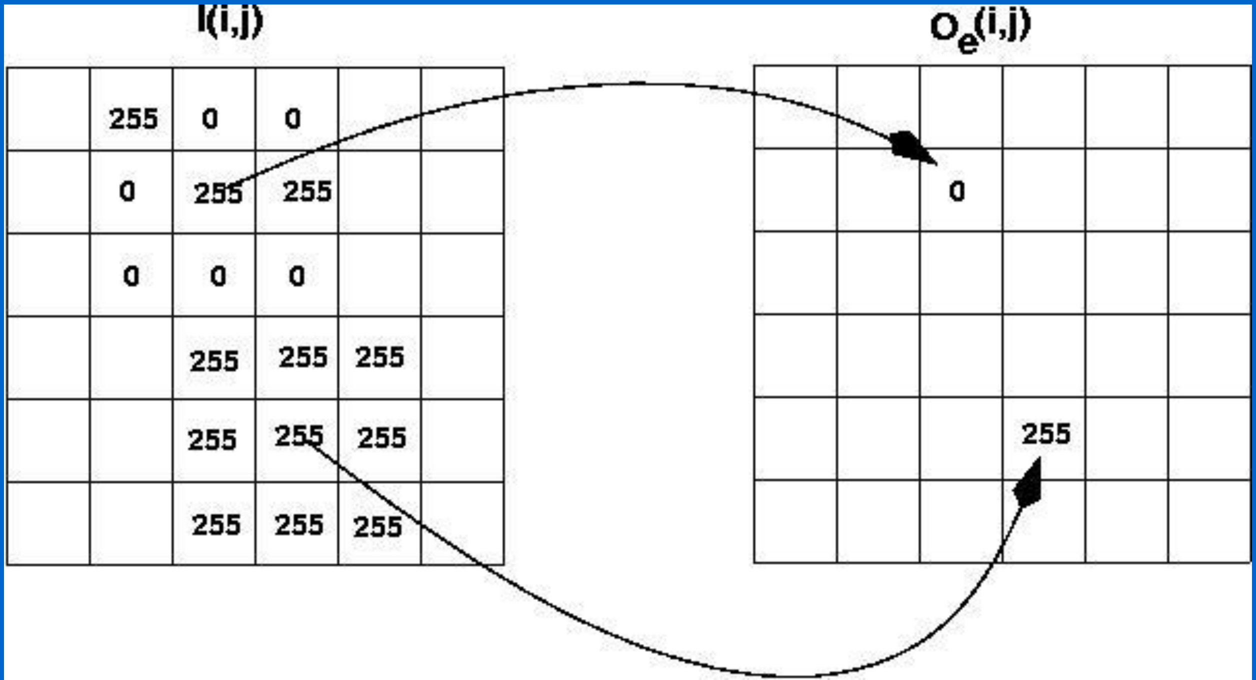


dilated



erode(image) -- client function

$$O_e(i,j) = \begin{cases} 255 & \textit{if} \quad \text{all 8 neighbors are 255} \\ I(i,j) & \textit{if} \quad \text{at least one neighbor is 0} \end{cases}$$



I(i,j)

	255	0	0		
	0	255	255		
	0	0	0		
		255	255	255	
		255	255	255	
		255	255	255	

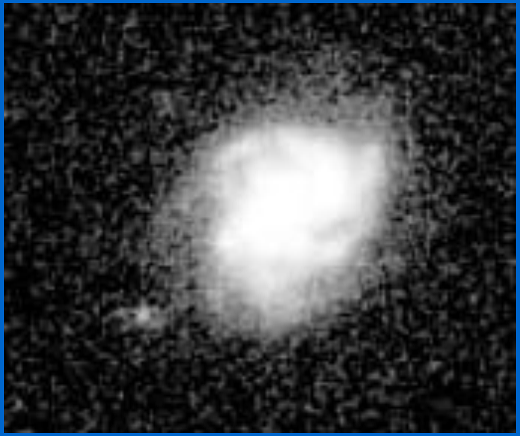
$O_e(i,j)$

		0			
			255		

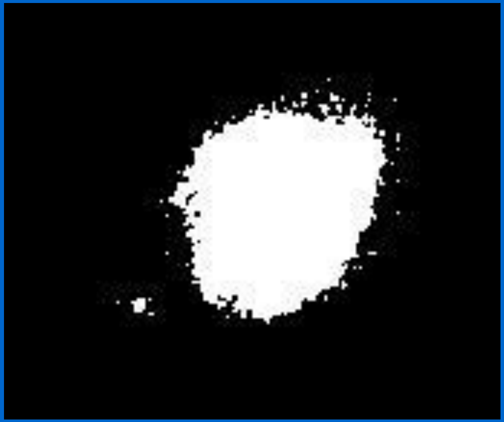
erode(image)

- Erosion “shrinks” the regions (i.e., removes a layer of boundary pixels)

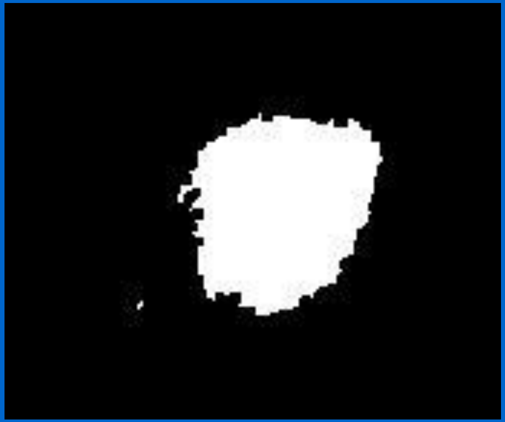
original



thresholded



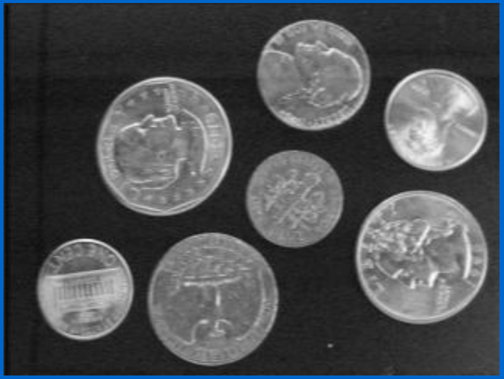
eroded

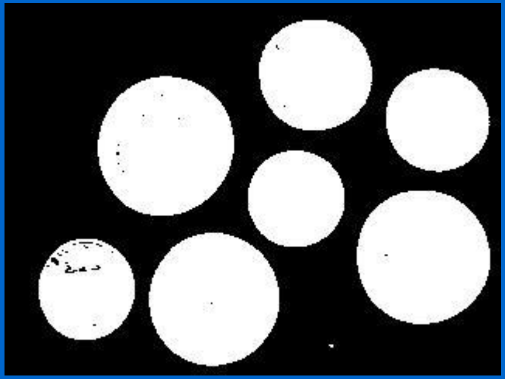


Filling in the holes of regions

- Apply dilation to fill in the holes.
- Apply erosion to restore the size of the regions.

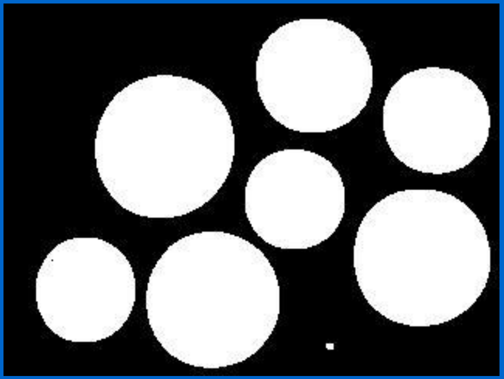
original

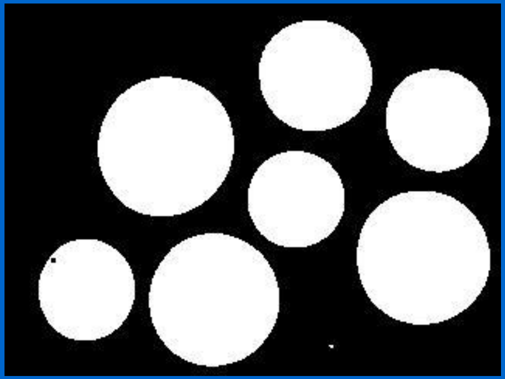




thresholded

dilated

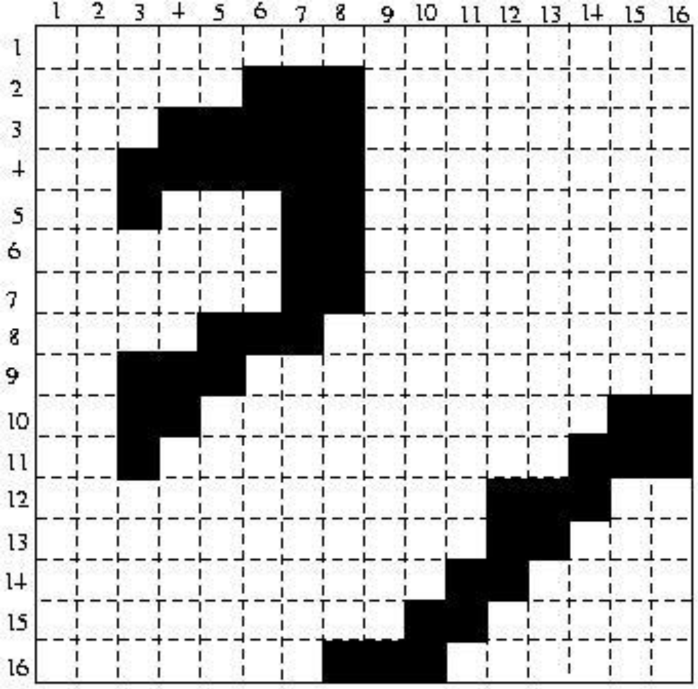


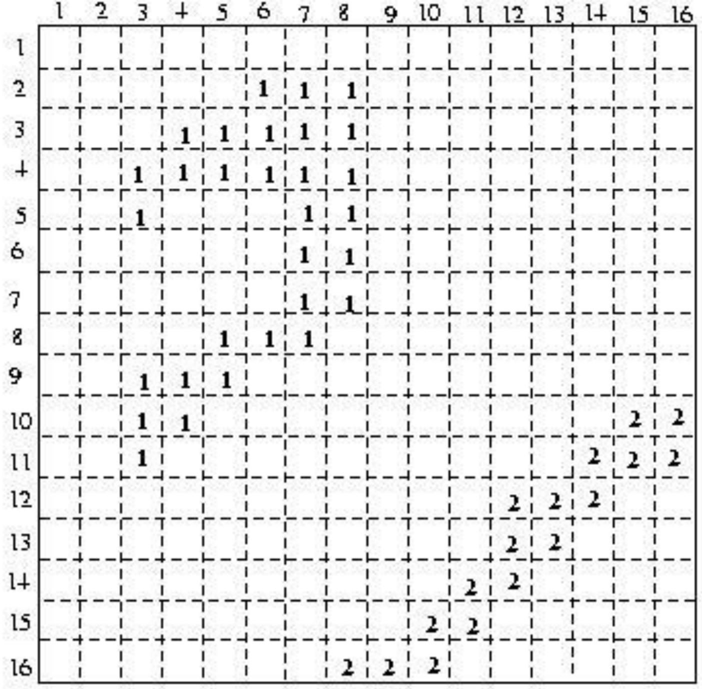


eroded

Connected Components Algorithm

- Finds the connected components in an image and assigns a unique label to all the points in the same component.



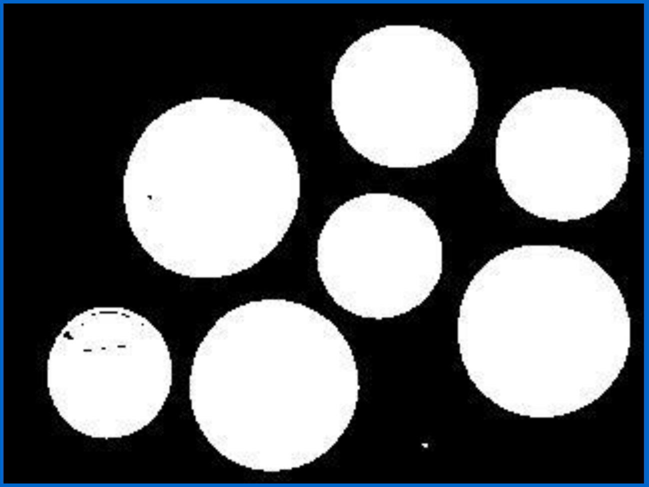


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1																
2						1	1	1								
3				1	1	1	1	1								
4			1	1	1	1	1	1								
5			1				1	1								
6							1	1								
7							1	1								
8					1	1	1									
9			1	1	1											
10			1	1											2	2
11			1											2	2	2
12												2	2	2		
13												2	2			
14											2	2				
15										2	2					
16								2	2	2						

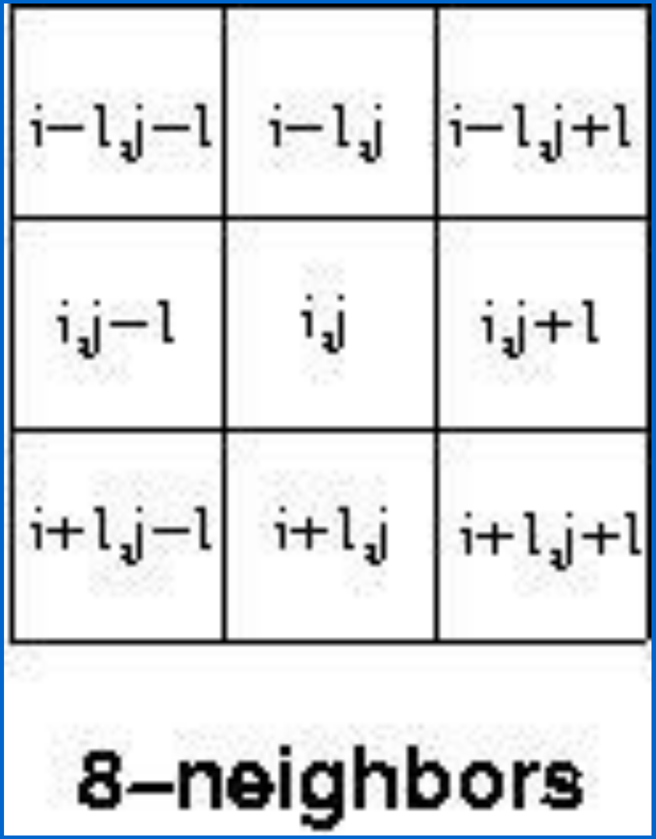
Connected Components Algorithm (cont'd)

1. Scan the thresholded image to find an unlabeled white (255) pixel and assign it a new label L.
2. Recursively assign the label L to all of its 255 neighbors.
3. Stop if there are no more unlabeled 255 pixels.
4. Go to step 1

Print number of regions found.



8-neighbors of (i,j)



i−1,j−1	i−1,j	i−1,j+1
i,j−1	i,j	i,j+1
i+1,j−1	i+1,j	i+1,j+1

8–neighbors

(client function)

int connectedComponents(inputImage, outputImage)

```
set outputImage --> 255 (white) // initialization
connComp=0;

for (i=0; i<N; i++)
  for(j=0; j<M; j++)
     if(inputImage[i][j] == 255 && outputImage[i][j]==255) {
       ++connComp;
        label = connComp;  // new label
        findComponent( parameters ); // recursive function

      // non-recursive functions
     //  findComponentDFS(inputImage, outputImage, i, j, label);
     //  findComponentBFS(inputImage, outputImage, i, j, label);
    }
return connComp;
```

findComponent(parameters)

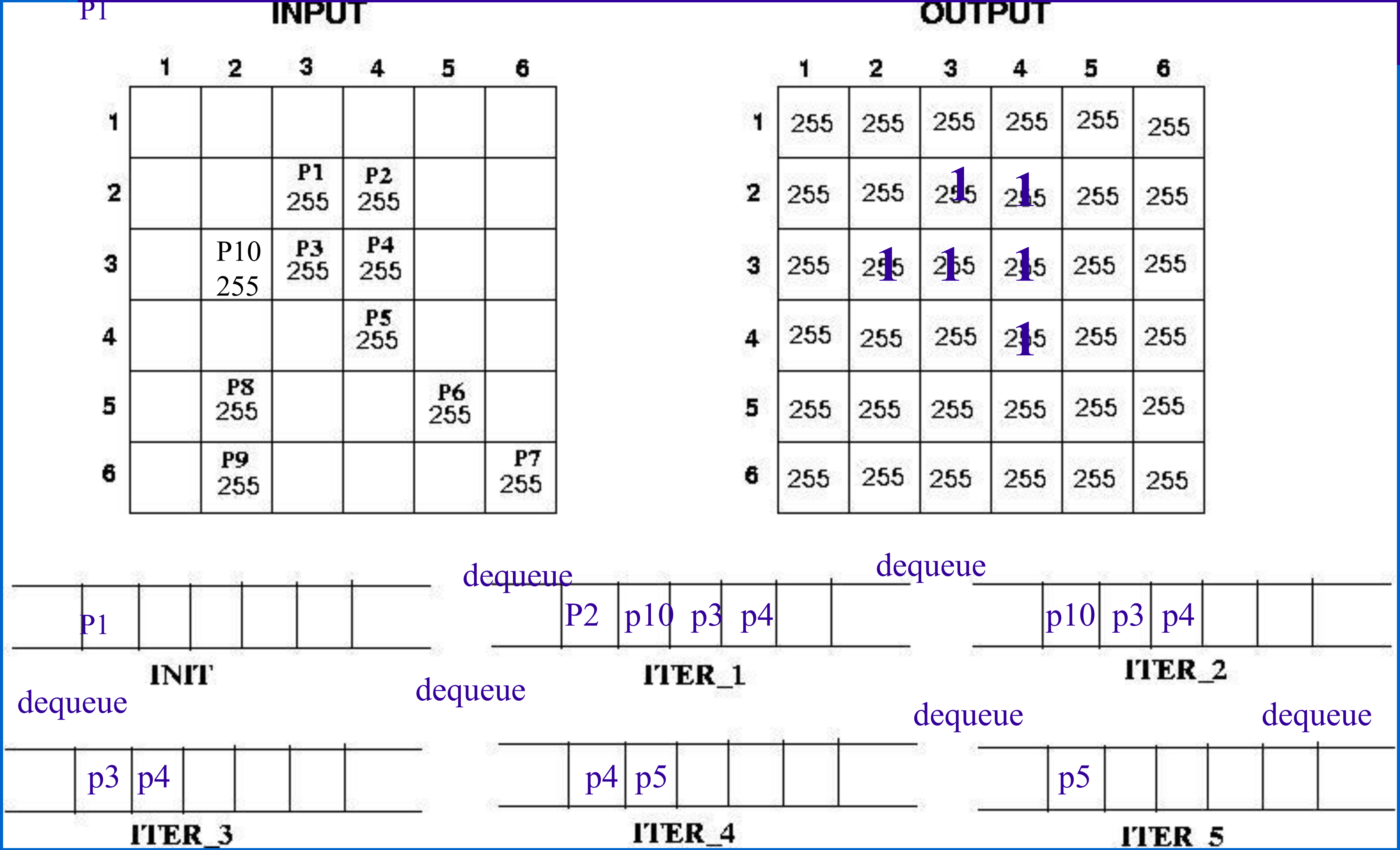
- Implement this as a recursive function.
- Think what the parameter list should be ...

Breadth-First-Search (BFS)

- The main structure used used by BFS is the queue.
- BFS uses a queue to “remember” the neighbors of pixel (i,j) that need to be labeled in future iterations.
- The closest neighbors of (i,j) are labeled first.
- BFS will first label all pixels at distance 1 from (i,j), then at distance 2, 3, etc.

findComponentBFS(inputImage, outputImage, i, j, label)

```
Queue.MakeEmpty();

Queue.Enqueue((i,j)); // initialize queue

while(!Queue.IsEmpty()) {
   Queue.Dequeue((pi,pj));
    outputImage[pi][pj] = label;
    for each neighbor (ni,nj) of (pi,pj) // push neighbors
        if(inputImage[ni][nj] == inputImage[pi][pj] && outputImage[ni][nj] == 255)
{
            outputImage[ni][nj] = -1; // mark this pixel
            Queue.Enqueue((ni,nj));
        }
  }
```



P1

INPUT

	1	2	3	4	5	6
1						
2			P1 255	P2 255		
3		P10 255	P3 255	P4 255		
4				P5 255		
5		P8 255			P6 255	
6		P9 255				P7 255

OUTPUT

	1	2	3	4	5	6
1	255	255	255	255	255	255
2	255	255	255 1	255 1	255	255
3	255	255 1	255 1	255 1	255	255
4	255	255	255	255 1	255	255
5	255	255	255	255	255	255
6	255	255	255	255	255	255

P1

INIT

dequeue

P2 p10 p3 p4

ITER_1

dequeue

p10 p3 p4

ITER_2

dequeue

p3 p4

ITER_3

dequeue

p4 p5

ITER_4

dequeue

p5

ITER_5

dequeue

Depth-First-Search (DFS)

- The main structure used used by DFS is the stack.
- DFS uses a stack to “remember” the neighbors of pixel (i,j) that need to be labeled in future iterations.
- The most recently visited pixels are visited first (i.e., not the closest neighbors)
- DFS follows a path as deep as possible in the image.
- When a path ends, DFS backtracks to the most recently visited pixel.

findComponentDFS(inputImage, outputImage, i, j, label)

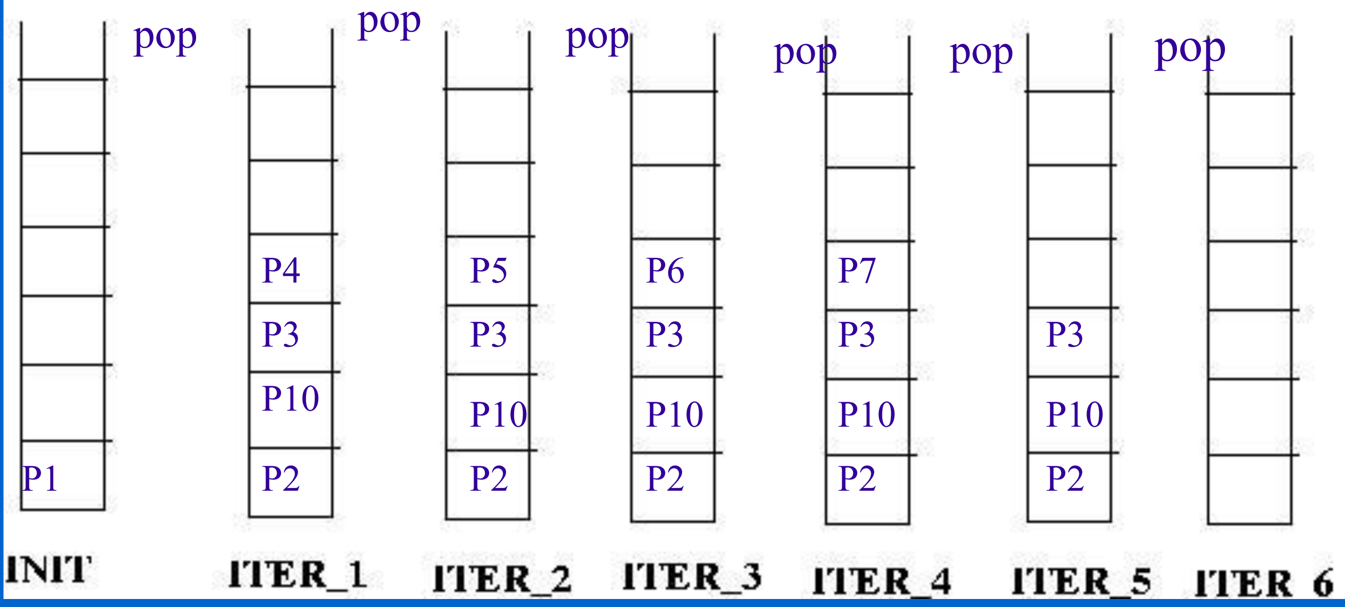
```
Stack.MakeEmpty();

Stack.Push((i,j)); // initialize stack

while(!Stack.IsEmpty()) {
   Stack.Pop((pi,pj));
    outputImage[pi][pj] = label;
    for each neighbor (ni,nj) of (pi,pj) // push neighbors
        if(inputImage[ni][nj] == inputImage[pi][pj] && outputImage[ni][nj] == 255)
{
            outputImage[ni][nj] = -1; // mark this pixel
            Stack.Push((ni,nj));
        }
  }
```

INPUT

	1	2	3	4	5	6
1						
2			P1 255	P2 255		
3		P10 255	P3 255	P4 255		
4				P5 255		
5		P8 255			P6 255	
6		P9 255				P7 255

OUTPUT

	1	2	3	4	5	6
1	255	255	255	255	255	255
2	255	255	1 255	255	255	255
3	255	255	1 255	1 255	255	255
4	255	255	255	1 255	255	255
5	255	255	255	255	1 255	255
6	255	255	255	255	255	1 255



INIT: P1 — pop

ITER_1: P4, P3, P10, P2 — pop

ITER_2: P5, P3, P10, P2 — pop

ITER_3: P6, P3, P10, P2 — pop

ITER_4: P7, P3, P10, P2 — pop

ITER_5: P3, P10, P2 — pop

ITER 6

etc.