

# **Module 2**

## **Binary Image Processing**

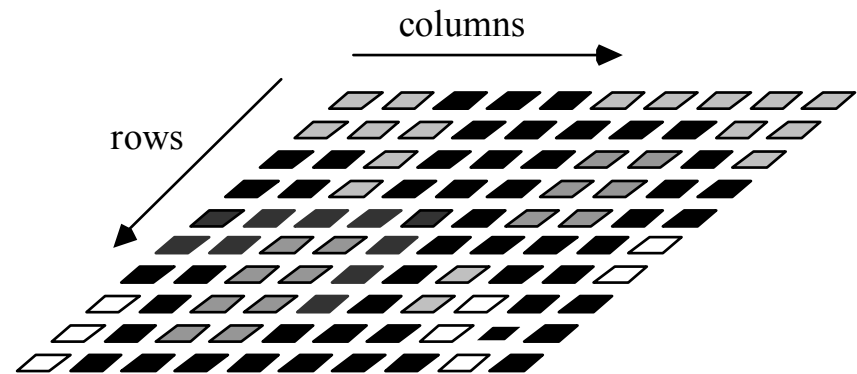
- **Binary Image Generation**
- **Logical Operations**
- **Blob Coloring**
- **Binary Morphology**
- **Binary Template Matching**
- **Binary Image Compression**

# Binary Images

- A **digital image** is an array of numbers: sampled image intensities:

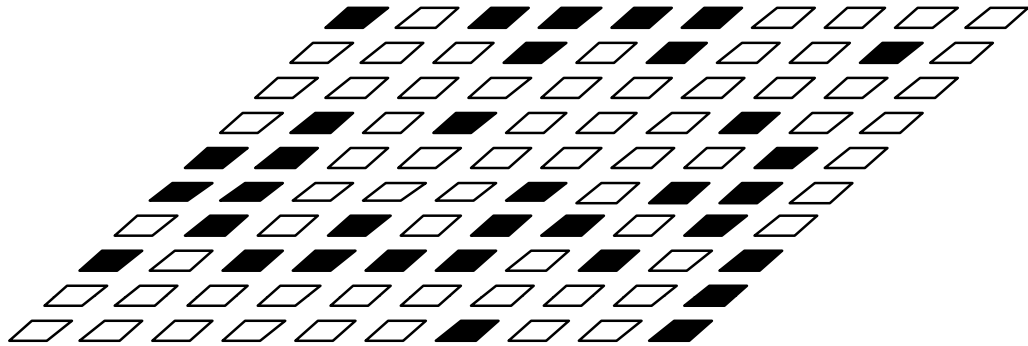
A 10 x 10 image

- Each **gray level** is **quantized**: assigned one of a finite set of numbers  $[0, \dots, K-1]$ .
- $K = 2^B$  possible gray levels: each pixel represented by  $B$  bits.



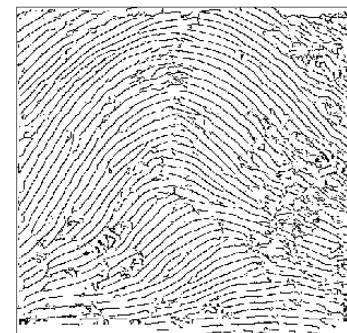
- **Binary images** have  $B = 1$ .

# Binary Images



A 10 x 10 binary image

- In **binary images** the (logical) values '0' and '1' often indicate the absence/presence of an image property in an associated gray-level image:
  - High vs. low intensity (brightness)
  - Presence vs. absence of an object
  - Presence vs. absence of a property



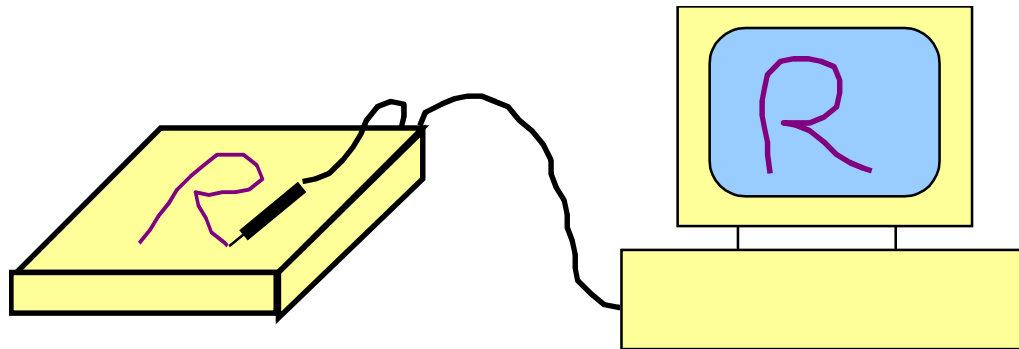
Example: Presence of fingerprint ridges

# **Display Convention for Binary Images**

- For printed page (tests and most of the notes):
  - Unless otherwise stated, BLACK = 1 = TRUE and WHITE = 0 = FALSE
  - Why? Usually the “objects” are small compared to the background and this creates a “cleaner” look in print. Think of a page in a book: the text is printed in black on a white background.
- For the computer monitor (and most homework):
  - Unless otherwise stated, WHITE = 1 = TRUE = 255d and BLACK = 0 = FALSE = 0d.
  - This creates a more “natural” look on the monitor.

# Binary Image Generation

- Binary images can come from **simple sensors** with binary output, e.g., **tablet, resistive pad, or light pen.**



- Quite useful for **engineering drawing, entering handprinted characters, etc.**

## Binary Images From Gray-Level Images

- Often, a binary image is obtained by **thresholding** a gray-level image.
- Why do this?
  - 8-fold reduction in required storage
  - Simple abstraction of information
  - Fast processing - **logical operators**
  - Good binary compression algorithms

# Simple Thresholding

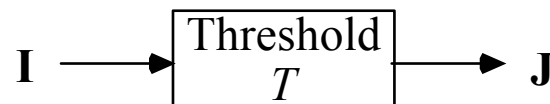
- The simplest image processing operation.
- An extreme form of **quantization**.
- Requires the definition of a **threshold**  $T$  (that falls in the gray-scale range).
- **Every** pixel intensity is compared to  $T$ , and a binary decision rendered.
- Ex: binary pixel = TRUE if gray-level pixel  $> T$

# Simple Thresholding

- Suppose **gray-level** image **I** has  
 $K$  gray-levels:  $0, 1, \dots, K-1$
- Select a threshold  $0 \leq T \leq K-1$ .
- Compare **every** gray-level in **I** to  $T$ .
- Define a new **binary image J** as follows:

$$J(m, n) = '1' \text{ if } I(m, n) \geq T$$

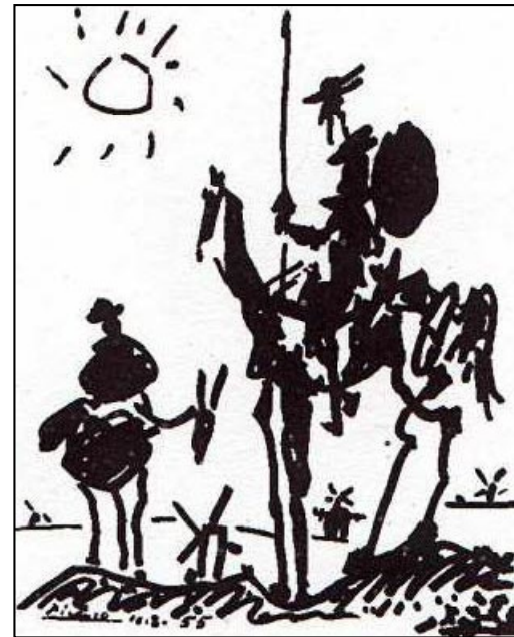
$$J(m, n) = '0' \text{ if } I(m, n) < T$$





# Binary Image Information

- Artists have long understood that binary images contain much information: form, structure, shape, etc.



“Don Quixote” by Pablo Picasso

BLACK = TRUE

WHITE = FALSE

original image,  $B=8$



$T=64$



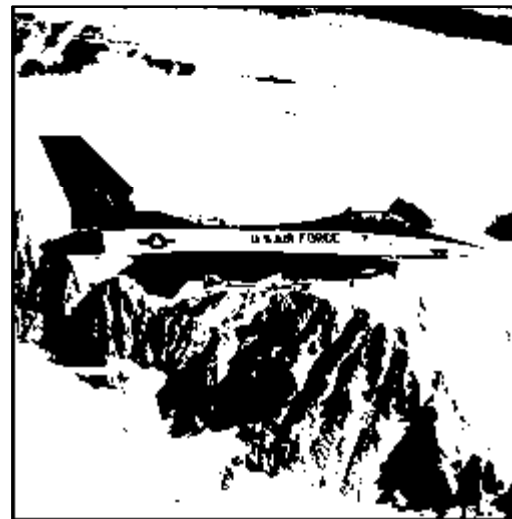
WHITE = TRUE

BLACK = FALSE

$T=128$



$T=192$



original image,  $B=8$



$T=32$



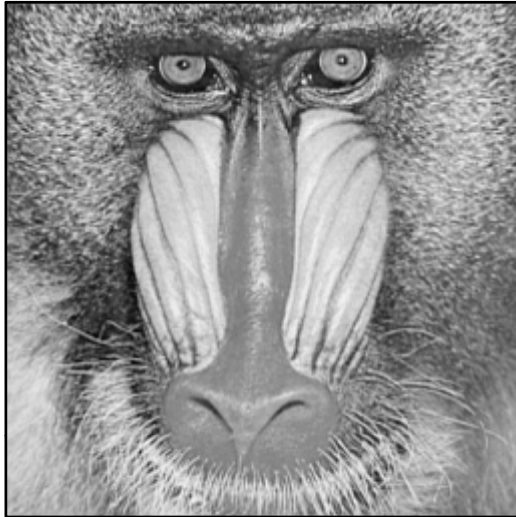
$T=128$



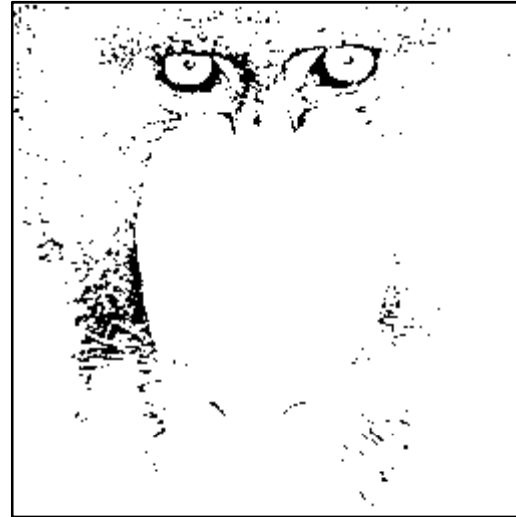
$T=192$



original image,  $B=8$



$T=64$



$T=128$



$T=160$



original



$T=32$



$T=64$



$T=96$



$T=128$



$T=160$



$T=192$



$T=224$



# Threshold Selection

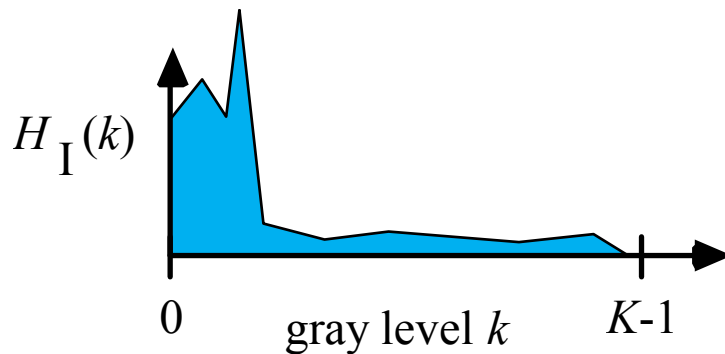
- The usefulness of the **binary image J** obtained by thresholding **I** depends heavily on the **threshold  $T$** .
- Different thresholds may give different valuable abstractions of the image (preceding examples).
- Some images do not produce any interesting results when thresholded by any  $T$ .
- So: How does one decide if thresholding is possible? How does one select the threshold  $T$ ?

# Gray-Level Image Histogram

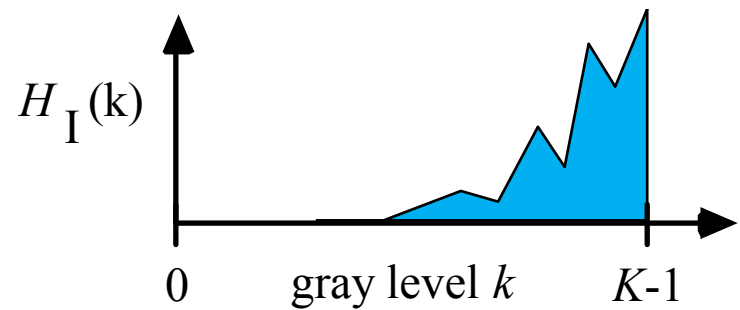
- The **histogram**  $H_I$  of image **I** is a **graph** of the **frequency of occurrence** of each gray level in **I**.
- $H_I$  is a one-dimensional function (array) with domain  $0, \dots, K-1$ .
- $H_I(k) = n$  if **I** contains **exactly**  $n$  pixels with gray level  $k$ ;  $k = 0, \dots, K-1$ .

# Histogram Appearance

- The **appearance** of a histogram can tell you a lot about the image:



Predominantly dark image



Predominantly light image

- Could be histograms of **underexposed** and **overexposed** images, respectively.

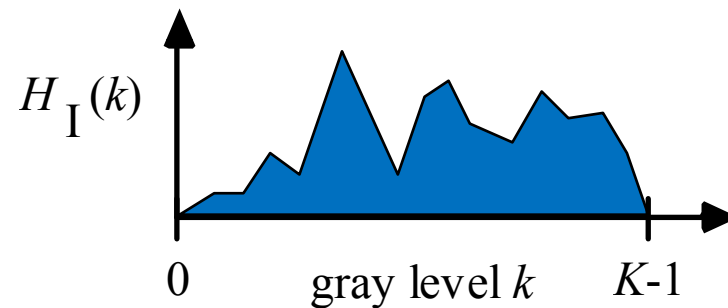


# Histogram Information

- The histogram contains first-order information only. It **can** tell you:
  - The average, min, and max pixel values, as well as the standard deviation.
  - The probability that a pixel picked at random from the image has value 32.
- But it does **not** contain any 2<sup>nd</sup> or higher-order information ( i.e., how the pixel values occur together in groups of two or more).
  - It can tell you variance, but it cannot tell you covariance.
  - It can't tell you the probability that a pixel with value 32 occurs next to a pixel with value 40.
  - If you take an image and scramble it up by moving all the pixels around to make a mess, the histogram remains **unchanged**.

# Histogram Appearance

- Here is a well-distributed histogram:

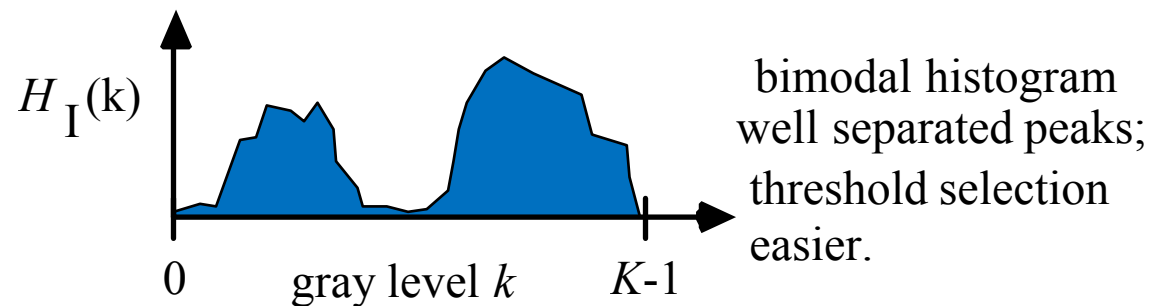
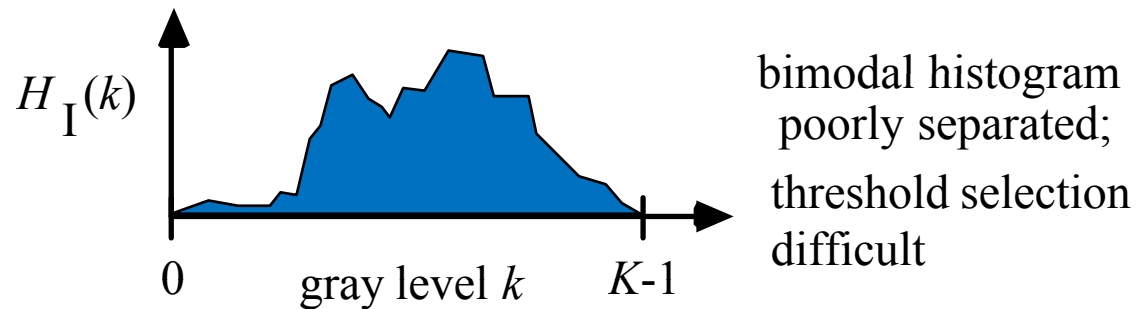


- It could be from an image that is neither too bright nor too dark.

# Bimodal Histogram

- Thresholding usually works best when there are **dark objects** on a **bright background**.
- Or when there are **bright objects** on a **dark background**.
- Images of this type tend to have histograms with **distinct peaks or modes**.
- Threshold selection is easier when the peaks are well separated.

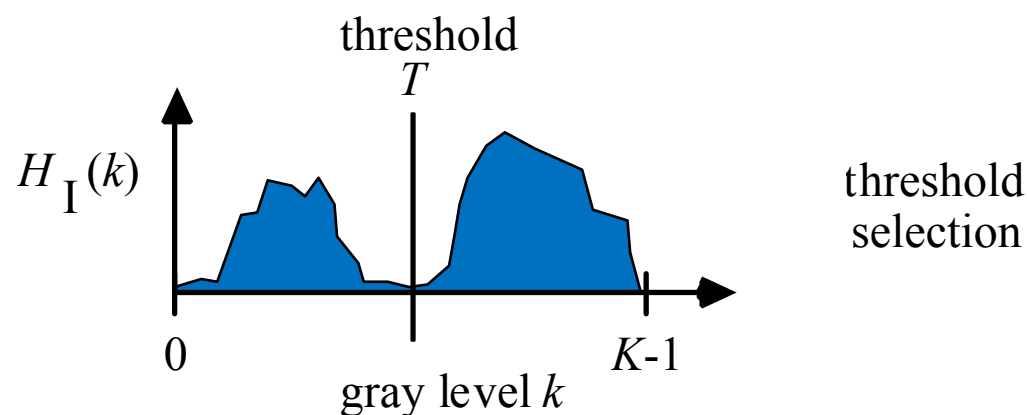
# Bimodal Histogram



- Often, the peaks (modes) correspond to the object(s) and to the background.

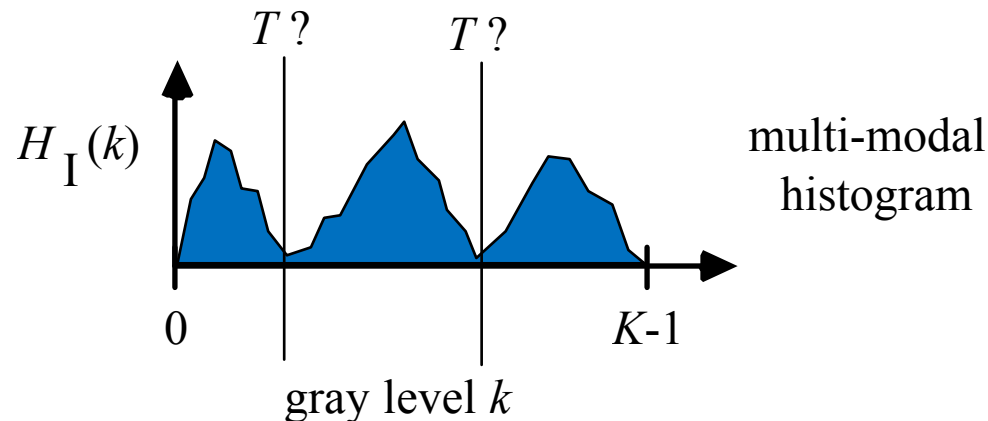
# Threshold Selection for Bimodal Histogram

- Suppose the histogram is bimodal and well separated.
- Placing the threshold  $T$  between the modes can often yield a binary image that is TRUE on the object and FALSE on the background (or vice-versa).
- The **exact optimal** threshold value is usually difficult to determine.



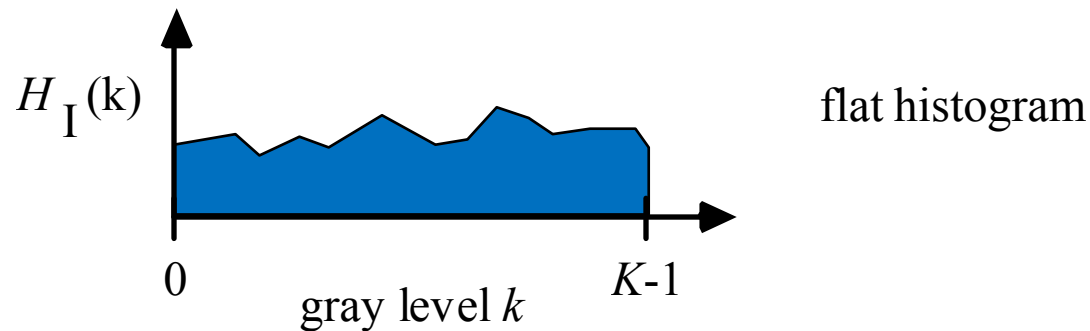
# Multi-Modal Histogram

- The histogram may have **multiple modes**. Different values for  $T$  will give very different results.
- In some cases, the multiple modes may be from multiple objects.
  - Placing multiple thresholds between the modes may separate out the objects in some cases.
  - This is called “multi-thresholding.”



# Flat Histogram

- The histogram may be "flat," making thresholding nearly impossible:



# Discussion of Histogram Types

- We'll use the histogram extensively for **gray-level processing** later.
- Some general observations for now:
  - **Bimodal histograms** often imply **objects** and **background** of different average brightness.
  - **Bimodal histograms** are easiest to threshold.
  - The ideal result is a simple binary image showing **object/background separation**
  - Ex: printed text characters, blood cells in solution, machine parts on an assembly line.



# Discussion of Histogram Types

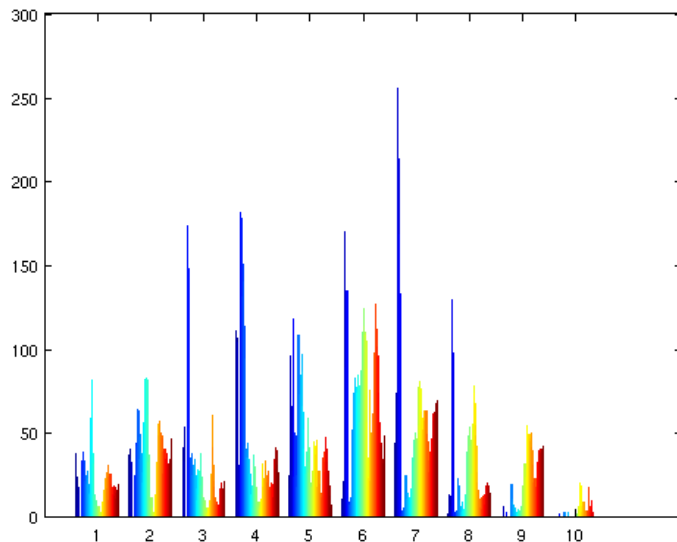
- **Multi-modal histograms** often occur in images of multiple objects with different average brightness.
- **“Flat” or level histograms** usually imply more complex images, containing detail, non-uniform background, etc.
- Thresholding rarely gives perfect results. Usually, some kind of **region correction** must be applied.
- We will study a region correction technique later in this module (blob coloring).

# A WARNING About Matlab

- Matlab provides a built-in function “hist” to compute histograms.
- But to use a Matlab built-in function correctly, you **MUST** read the documentation!

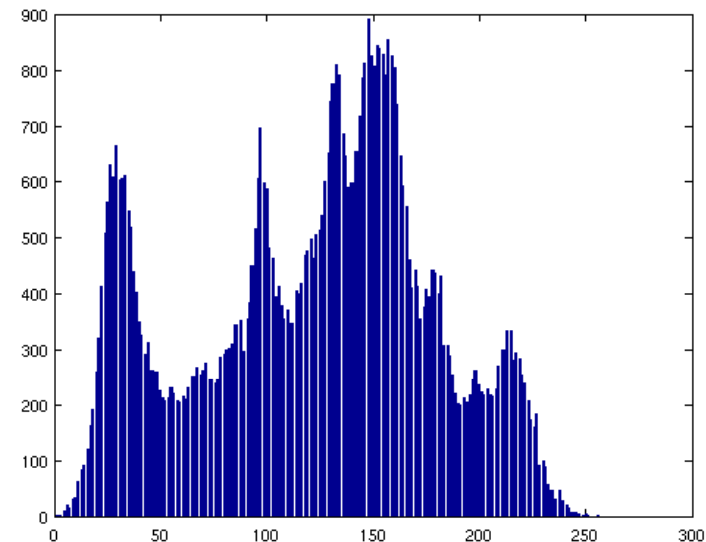
**Incorrect!!**

```
h = hist(Lena);  
bar(h);
```



**Correct:**

```
h = sum(hist(Lena,0:255)');  
bar(h);
```



# Making a Histogram in C

```
// the image is in unsigned char array x
// the total number of pixels is n
// the histogram will go into int array hist
// since K=256, hist has 256 bins
for (i=0; i<256; i++) {
    hist[i] = 0;
}
For (i=0; i<n; i++) {
    hist[x[i]]++;
}
```

# Thresholding in C and Matlab

In C:

```
// x is the input image
// y is the output image
// n is the number of pixels
// T is the threshold

for (i=0; i<n; i++) {
    if (x[i] < T) {
        y[i] = 0x00;
    } else {
        y[i] = 0xFF;
    }
}
```

In Matlab:

```
% x is the input image
% y is the output image

y = 255 * (x >= T);

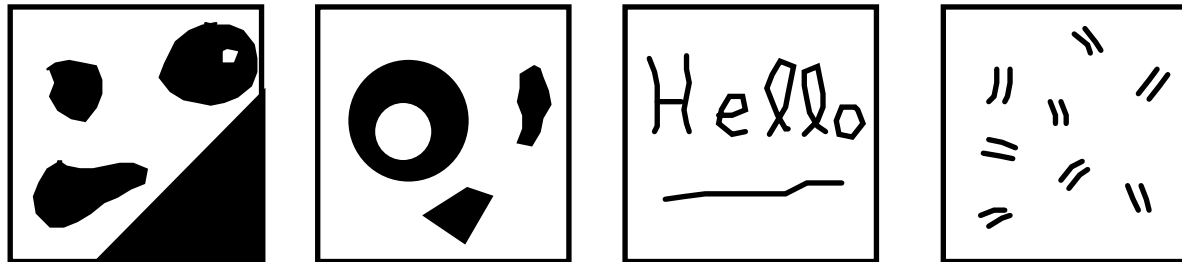
% done.

% can display like this:

figure(1);colormap(gray(256));
image(y);axis('image');
```

# Logical Operations on Binary Images

- Suppose we have obtained some binary images:



- Since the pixels have logical values (TRUE/FALSE), we can perform logical operations like NOT, AND, and OR on them.

# Logical Operations - Notation

- Suppose  $X_1, \dots, X_n$  are **binary variables**. For example, binary pixel values.
- Here is the notation we will use:

$$\text{NOT}(X_1) = \textbf{complement of } X_1$$

$$\text{AND}(X_1, X_2) = X_1 \wedge X_2$$

$$\text{OR}(X_1, X_2) = X_1 \vee X_2$$

## More Notation

- Multivariate logical operations:

$$\text{AND}(X_1, \dots, X_n) = X_1 \wedge \dots \wedge X_n$$

$$\text{OR}(X_1, \dots, X_n) = X_1 \vee \dots \vee X_n$$

- Binary Majority:

$\text{MAJ}(X_1, \dots, X_n) = '1'$  if more 1's than 0's.

Else  $\text{MAJ}(X_1, \dots, X_n) = '0'$ .

# Binary Majority

- Defined for an **odd** number of binary variables only.

| $X_1$ | $X_2$ | $X_3$ | $\text{MAJ}(X_1, X_2, X_3)$ |
|-------|-------|-------|-----------------------------|
| 0     | 0     | 0     | 0                           |
| 0     | 0     | 1     | 0                           |
| 0     | 1     | 0     | 0                           |
| 0     | 1     | 1     | 1                           |
| 1     | 0     | 0     | 0                           |
| 1     | 0     | 1     | 1                           |
| 1     | 1     | 0     | 1                           |
| 1     | 1     | 1     | 1                           |

Three-Variable  
Truth Table  
for MAJ

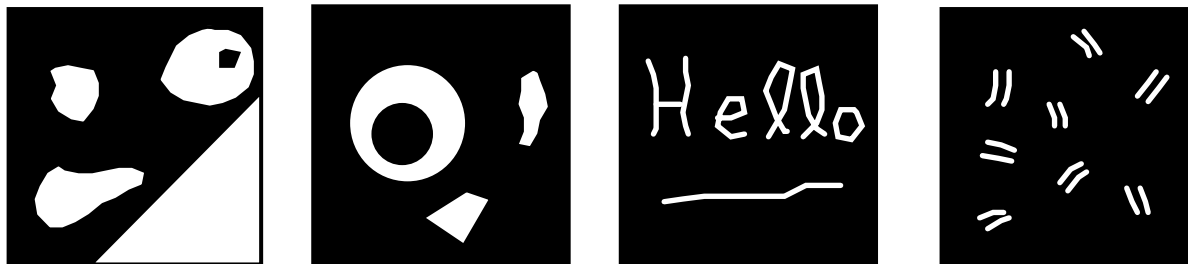
- We will define some very useful **binary image filters** using MAJ later on.



# Image Complement

- We will now define several **point-wise** logical image operations.
- **Image complement:**

$\mathbf{J} = \text{NOT}(\mathbf{I})$  if  $J(m, n) = \text{NOT}[I(m, n)] \quad \forall (m, n)$



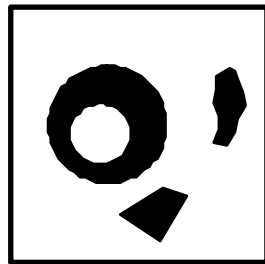
# Image AND

- Image **AND**:

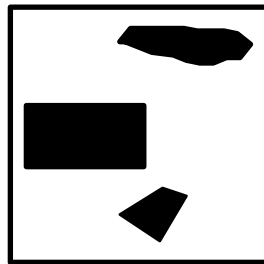
$$\mathbf{J} = \text{AND}(\mathbf{I}_1, \mathbf{I}_2) = \mathbf{I}_1 \wedge \mathbf{I}_2$$

$$\text{if } J(m, n) = \text{AND}[I_1(m, n), I_2(m, n)] \quad \forall (m, n)$$

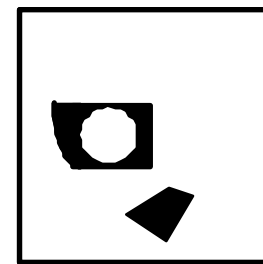
- Shows **overlap** between objects in  $\mathbf{I}_1$  and  $\mathbf{I}_2$ :



$\mathbf{I}_1$



$\mathbf{I}_2$



$\mathbf{J} = \mathbf{I}_1 \wedge \mathbf{I}_2$

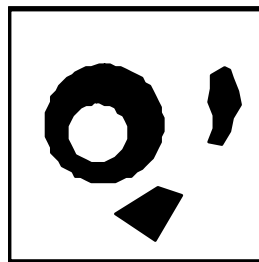
# Image OR

- Image **OR**:

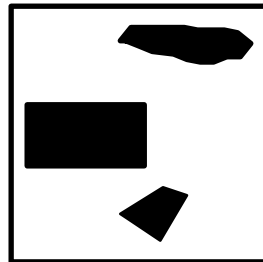
$$\mathbf{J} = \text{OR}(\mathbf{I}_1, \mathbf{I}_2) = \mathbf{I}_1 \vee \mathbf{I}_2$$

$$\text{if } J(m, n) = \text{OR}[I_1(m, n), I_2(m, n)] \quad \forall (m, n)$$

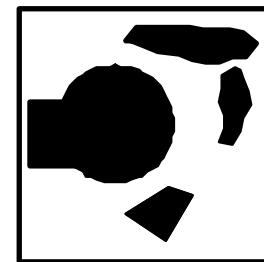
- Marks objects in  $\mathbf{I}_1$  or  $\mathbf{I}_2$  (or both)



$\mathbf{I}_1$



$\mathbf{I}_2$



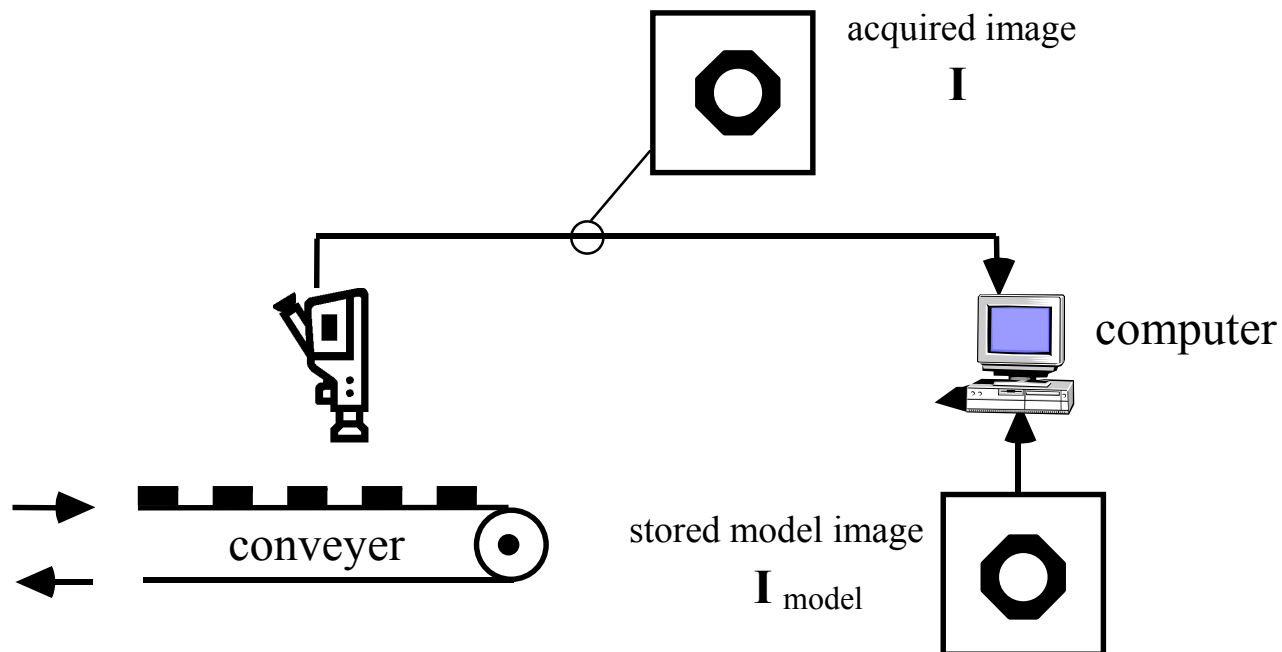
$\mathbf{J} = \mathbf{I}_1 \vee \mathbf{I}_2$

## Comments on AND, OR, MAJ

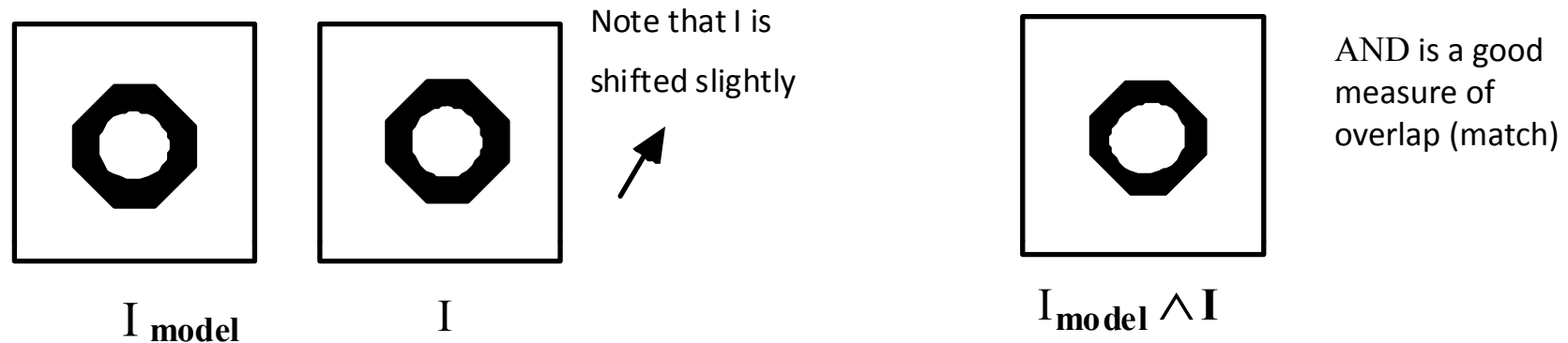
- As we will see later, AND, OR, and MAJ are **very** useful when applied to **local image regions** rather than the whole image.
- The usefulness of **whole-image** AND, OR and MAJ is quite limited.
- With a few exceptions ...

# Assembly Line Example

- An assembly line image inspection system.

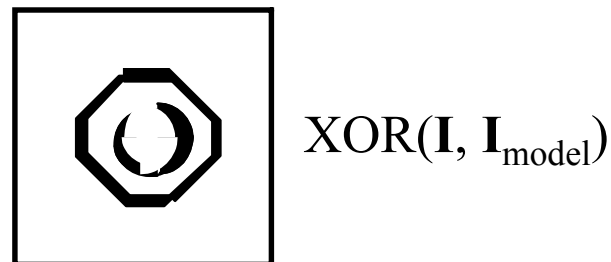


- **Objective**: Meaningfully compare the acquired image  $I$  to the stored image  $I_{\text{model}}$



- A good measurement of mismatch: **XOR**.

$$\text{XOR}(I, I_{\text{model}}) = \text{OR}\{\text{AND}[I_{\text{model}}, \text{NOT}(I)], \text{AND}[\text{NOT}(I_{\text{model}}), I]\}$$



- Not done yet...

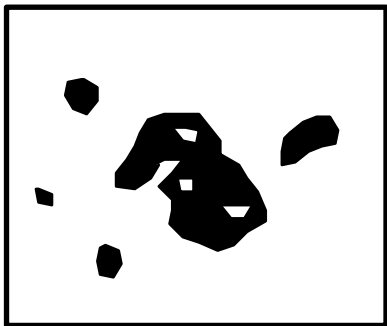
- XOR gives an image of **mismatch**.
- To decide if there is a problem or flaw, the ratio or percentage can be computed:

$$\text{PERCENT} = \frac{\text{\# of '1' pixels in XOR(I, I_{\text{model}})}}{\text{\# of '1' pixels in I_{\text{model}}}}$$

- This may be compared to a pre-determined tolerance percentage  $Q$ .
- If  $\text{PERCENT} > Q$ , then the part may be **flawed** or **incorrectly placed**.

# BLOB COLORING

- Also called “**Region Labeling**” and “**Connected Components Analysis**”
- **Motivation:** Gray-level image thresholding often produces an imperfect binary image; there will usually be extraneous blobs or holes (misclassified pixels) due to:
  - noise
  - overlapping object/background histograms
  - nonuniform object/background reflectance



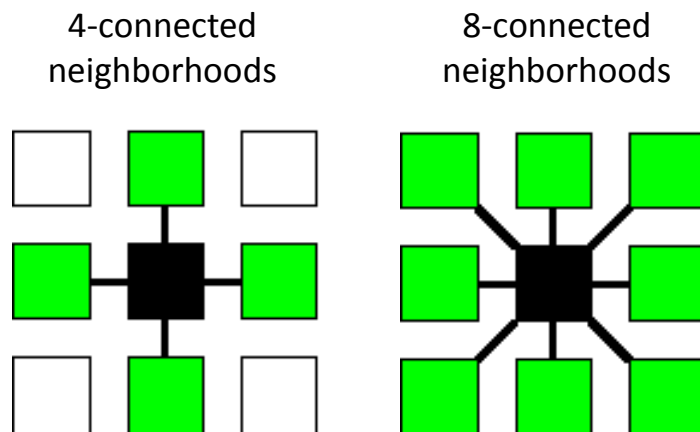
typical thresholded  
image result

- Blob coloring is a simple method for **labeling** (a.k.a. **coloring**) all the objects (blobs) in a binary image.



# What is a “Blob”?

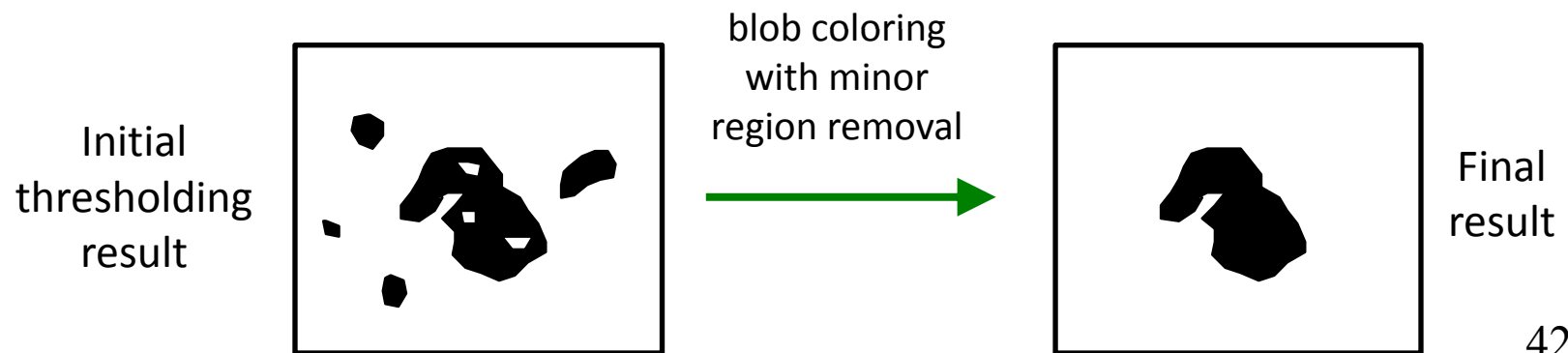
- In a binary image, a **blob** or **connected component** is a group of “1” pixels that are all “connected” to each other.
- This means that there is a connected path of “1” pixels between any two pixels in the blob.
- The notion of which pixels are “connected” to each other is defined by the **topology** that we impose on the image.
- The topology specifies which neighbor pixels are connected to each other and which ones aren’t.
- There are two common topologies:



Unless otherwise specified,  
assume the **4-connected**  
topology in this course

# Blob Coloring with Minor Region Removal





- In our present context, the assumption is that we have a grayscale image with one main object against a background.
- Our **goal** is to use thresholding to produce a binary image that is '1' on the object and '0' on the background.
- The thresholding result will generally be imperfect:
  - Some object pixels will be **misclassified** as background
  - Some background pixels will be **misclassified** as object
- Blob coloring with minor region removal is a simple technique to accomplish region classification and correction after thresholding.



# Blob Coloring with Minor Region Removal

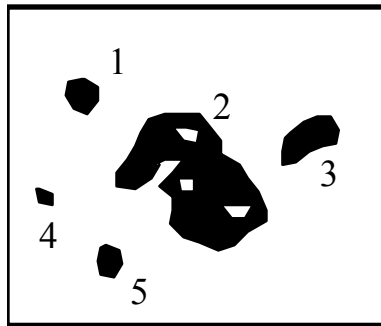
- Why might we want to do this?
  - It is often used as a “low level” image processing step that is performed prior to some “higher level” computer vision process.
  - The “higher level” process could be, e.g., target tracking, biometric person verification, autonomous vehicle/robot navigation, etc.
  - The higher level process will generally take both the binary image and the original grayscale image as input.
  - The binary image is used as a **map** that tells the higher level process where the object of interest is located in the grayscale image.
- How it works:
  1. Perform blob coloring to label the blobs.
  2. Perform blob counting to compute the size of each blob.
  3. Delete all but the largest blob (minor region removal).
  4. There will still be extraneous holes. So, invert and repeat steps 1-3.
  5. Invert again to get the final region corrected result.
- This will make more sense after we go through the details!

# Blob Coloring Algorithm

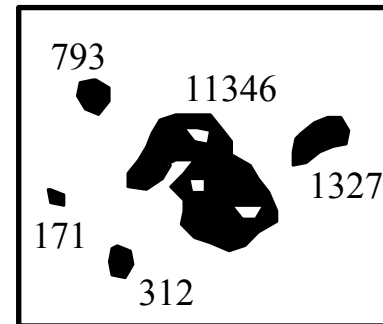
- For binary image **I**, define a "region color" array **R**.  
 $R(m, n) = \text{region number of pixel } I(m, n)$
- Set **R** = **0** (all zeros) and  $k = 1$  ( $k$  = counter for “next available region number”)
- While scanning the image left-to-right and top-to-bottom **do**
  - if  $I(m, n) = 1$  and  $I(m, n-1) = 0$  and  $I(m-1, n) = 0$  then // new blob found!  
 set  $R(m, n) = k$  and  $k = k + 1$ ; // use new color! 
  - if  $I(m, n) = 1$  and  $I(m, n-1) = 0$  and  $I(m-1, n) = 1$  then // part of an old blob  
 set  $R(m, n) = R(m-1, n)$ ; // use old color 
  - if  $I(m, n) = 1$  and  $I(m, n-1) = 1$  and  $I(m-1, n) = 0$  then // part of old blob  
 set  $R(m, n) = R(m, n-1)$ ; 
  - if  $I(m, n) = 1$  and  $I(m, n-1) = 1$  and  $I(m-1, n) = 1$  then   
 set  $R(m, n) = R(m-1, n)$ ; // all 3 are in the same blob!  
 if  $R(m, n-1) \neq R(m-1, n)$  then // oops! 2 old blobs are really the same!  
 Record that  $R(m, n-1)$  and  $R(m-1, n)$  are equivalent;

# Blob Coloring Example

- Distinct labels or "colors"  $k$  are assigned to each blob. Counting the pixels in each blob is simple (blob counting).



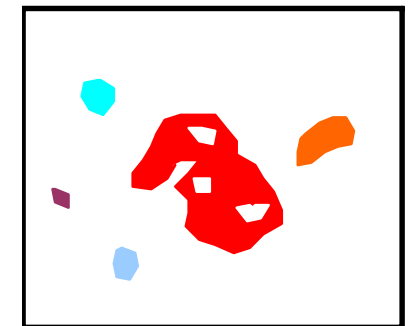
blob coloring  
result



blob counting  
result

- “Color” of largest blob: 2

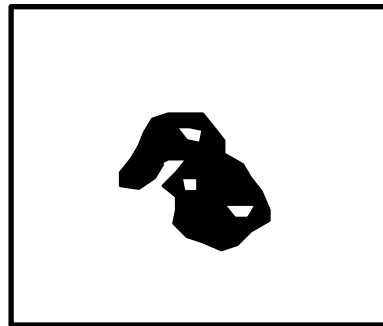
Colored blob display



# Minor Region Removal

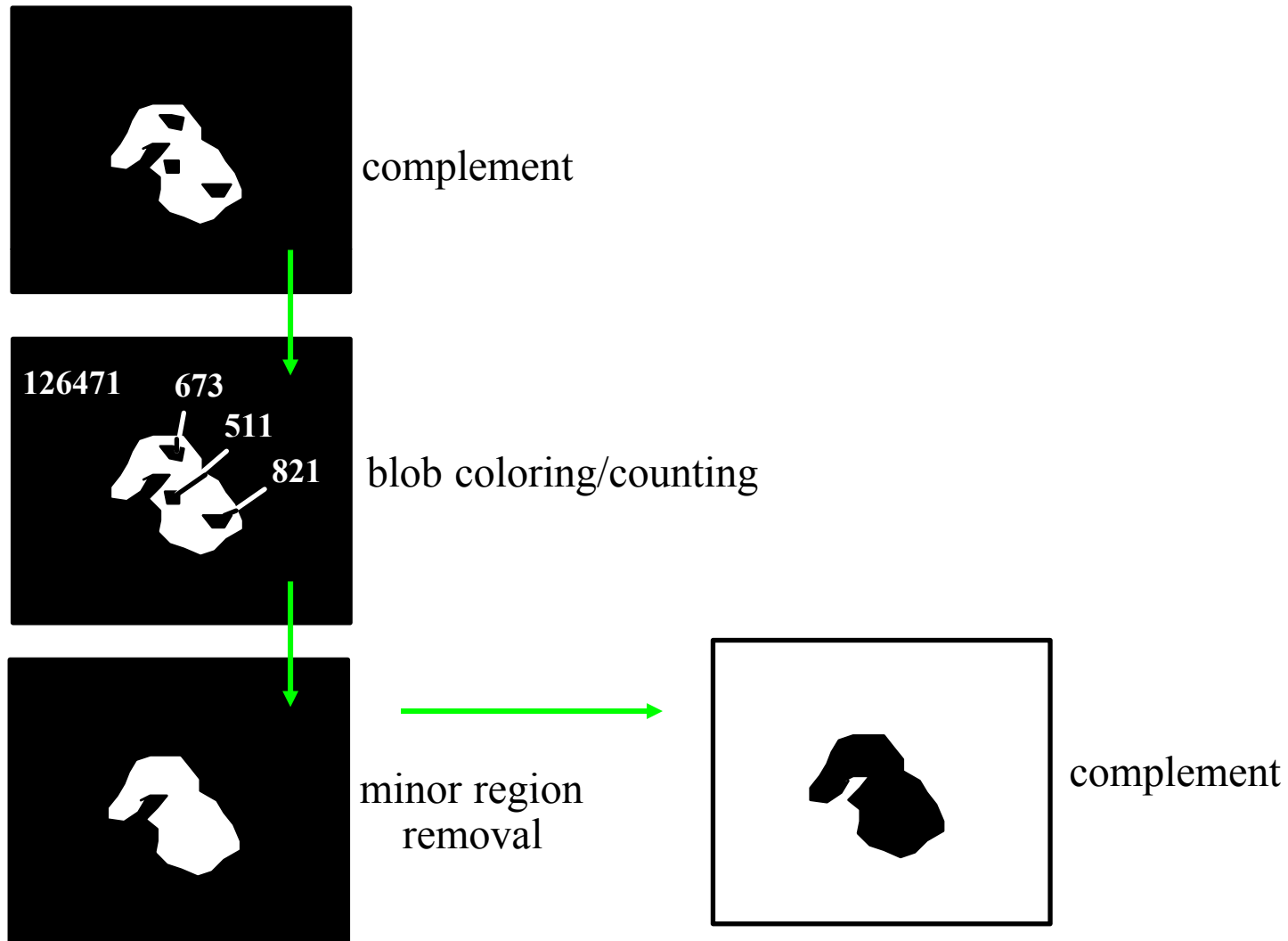
- Let  $q$  = "color" of largest region (blob)
- While scanning the image left-to-right and top-to-bottom **do**

if  $I(m, n) = 1$  and  $R(m, n) \neq q$  then  
set  $I(m, n) = 0$ ;



minor region  
removal

- Not finished! To remove the holes, we must repeat the procedure on the WHITE pixels:



# Binary Morphological Operations

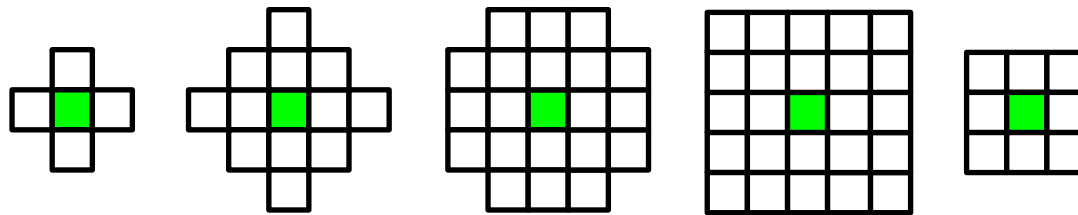
- The **most powerful** and **useful** class of binary image operators we will consider.
- The general framework is known as **mathematical morphology**  
**morphology = shape**
- Morphological operations affect the shapes of objects and regions in binary images.
- All processing done on a **local basis** - region or blob shapes are affected in a local manner.



- Morphological operators:
  - Expand (**dilate**) objects
  - Shrink (**erode**) objects
  - Smooth object boundaries and eliminate small regions or holes
  - Fill gaps and eliminate 'peninsulas'
- All is accomplished using **local logical operations**

# Structuring Elements

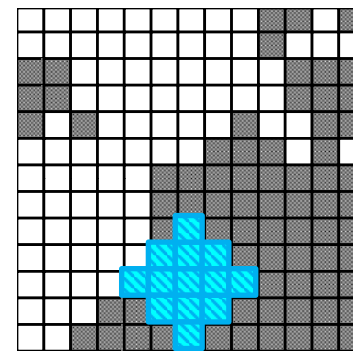
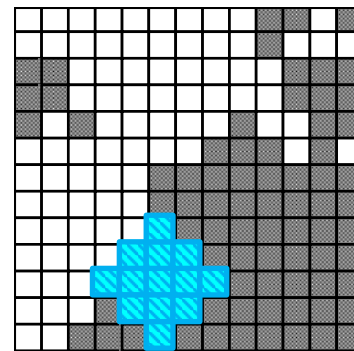
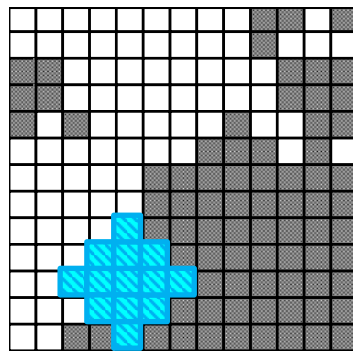
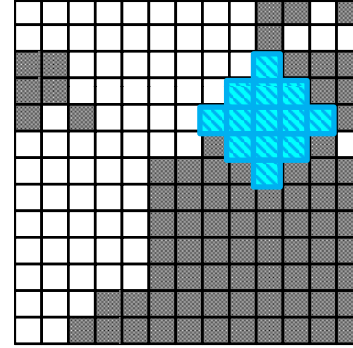
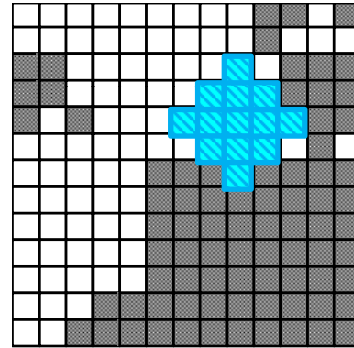
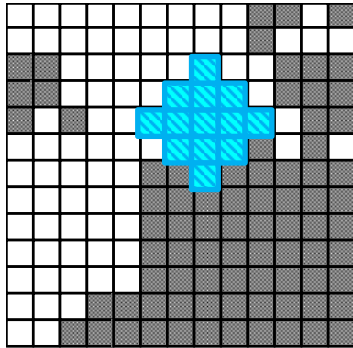
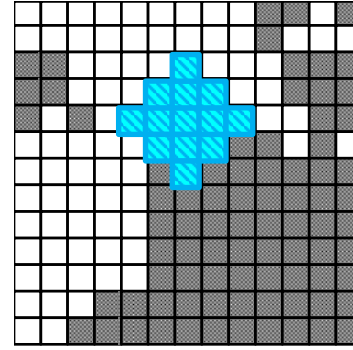
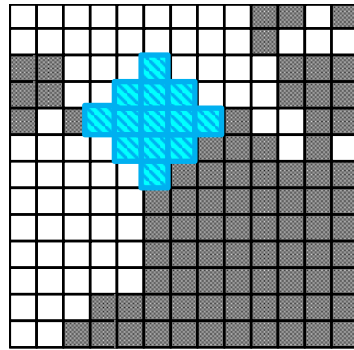
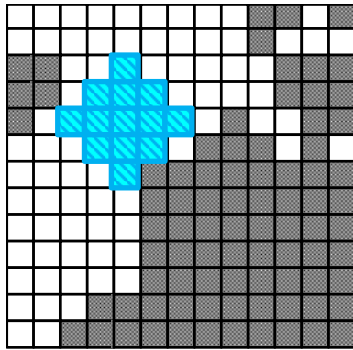
- Also known as “Windows”
- Some examples:



- A structuring element has a **reference point** (usually in the center – shown in green above)
- When the reference point is placed over a pixel in an image, a **set** of pixels are **selected** (the ones that are covered by the structuring element)

# Moving Window / Filtering Concept

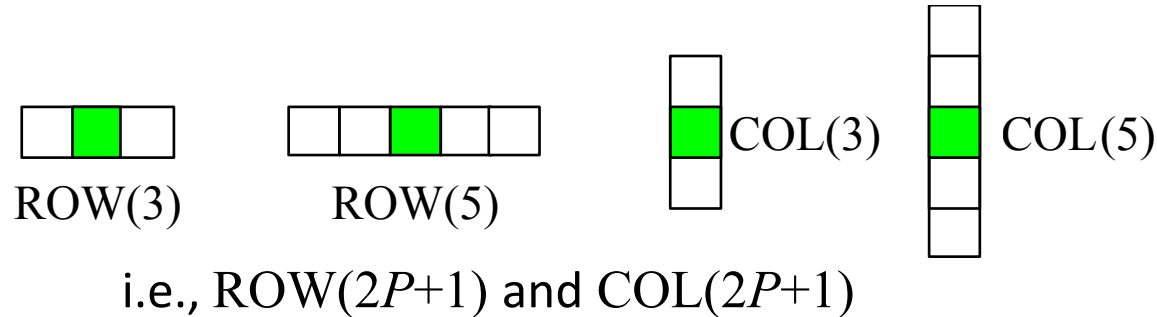
- To implement a morphological filter, the structuring element is passed over the image pixels, row-by row, column-by-column (moving window concept).
- Along the way, the structuring element is referenced to each pixel in the image, selecting a set of input pixels that are covered.
- A logical operation is performed on the selected set to produce a binary output pixel.
- **Usually**, the structuring element is approximately **circular** so that the filtering operation approximately commutes with object/image **rotation**.



A structuring  
element moving  
over an image.

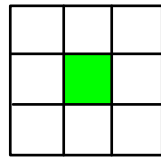
# Window Notation

- The terms 'window' and 'structuring element' are synonyms.
- In morphology, the term 'structuring element' is usually used.
- Some typical windows with a 1-D structure:

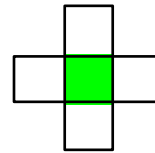


- The window usually covers an odd number of pixels ( $2P+1$ ) so that it is symmetric about the reference point and there is no phase shift between the input and output images.

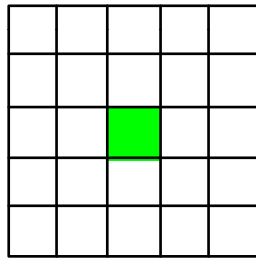
- Some windows with a 2-D structure:



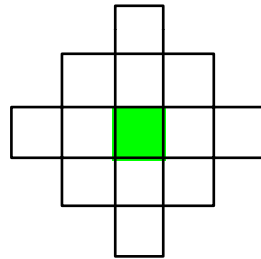
SQUARE(9)



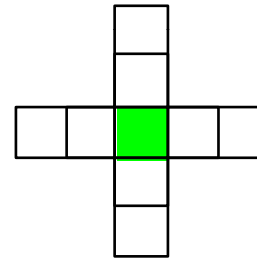
CROSS(5)



SQUARE(25)



CIRC(13)



CROSS(9)

SQUARE( $2P+1$ ), CROSS( $2P+1$ ), CIRC( $2P+1$ )

# Windowing Concept & “Phase”

- Consider a 1-D LTI system  $H_1$  with impulse response

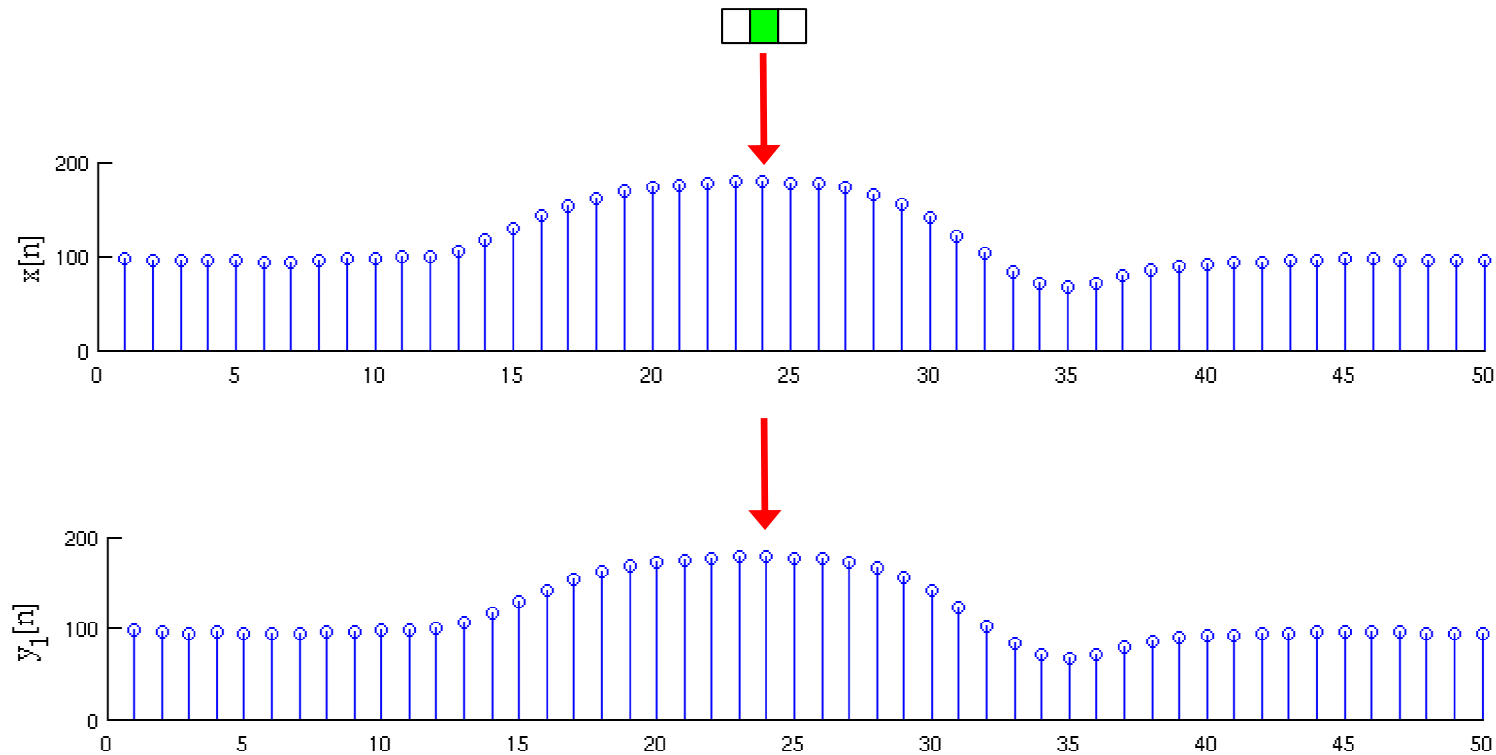
$$h_1[n] = \frac{1}{4}\delta[n+1] + \frac{1}{2}\delta[n] + \frac{1}{4}\delta[n-1].$$

- The output is

$$\begin{aligned}y_1[n] &= x[n] * h_1[n] \\&= \sum_{k=-\infty}^{\infty} h_1[k] x[n-k] \\&= \frac{1}{4}x[n+1] + \frac{1}{2}x[n] + \frac{1}{4}x[n-1]\end{aligned}$$

- Thus, at each point  $n$ , the number  $y_1[n]$  is a weighted average of three surrounding inputs that is *centered*.
- Note that  $H_1$  is not causal, since the current output  $y_1[n]$  depends on the future input  $x[n+1]$ .

- You can think of the filter  $H_1$  as having a 3-point window that slides over the input signal:



- Because the window is centered, there is no shift in the output signal relative to the input signal; i.e., the filter introduces no *delay*.



- Because  $h_1[n]$  is *real* and *even*, the frequency response  $H_1(e^{j\omega})$  is also *real* and *even*.
- The frequency response is given by

$$\begin{aligned}
 H_1(e^{j\omega}) &= \sum_{n=-\infty}^{\infty} h_1[n]e^{-j\omega n} \\
 &= \frac{1}{4}e^{j\omega} + \frac{1}{2} + \frac{1}{4}e^{-j\omega} \\
 &= \frac{1}{2} + \frac{1}{2}\cos \omega.
 \end{aligned}$$

- The phase response is  $\arg H_1(e^{j\omega}) = 0$ .
- This again shows that, because the window is centered, there is no shift (delay) between the input signal and the output signal.
- But the filter is not causal.
- To implement the filter in real-time, you would have to *shift*  $h_1[n]$  to the right by one sample to make it causal.

- So, now consider a 2<sup>nd</sup> 1-D LTI system  $H_2$  with impulse response

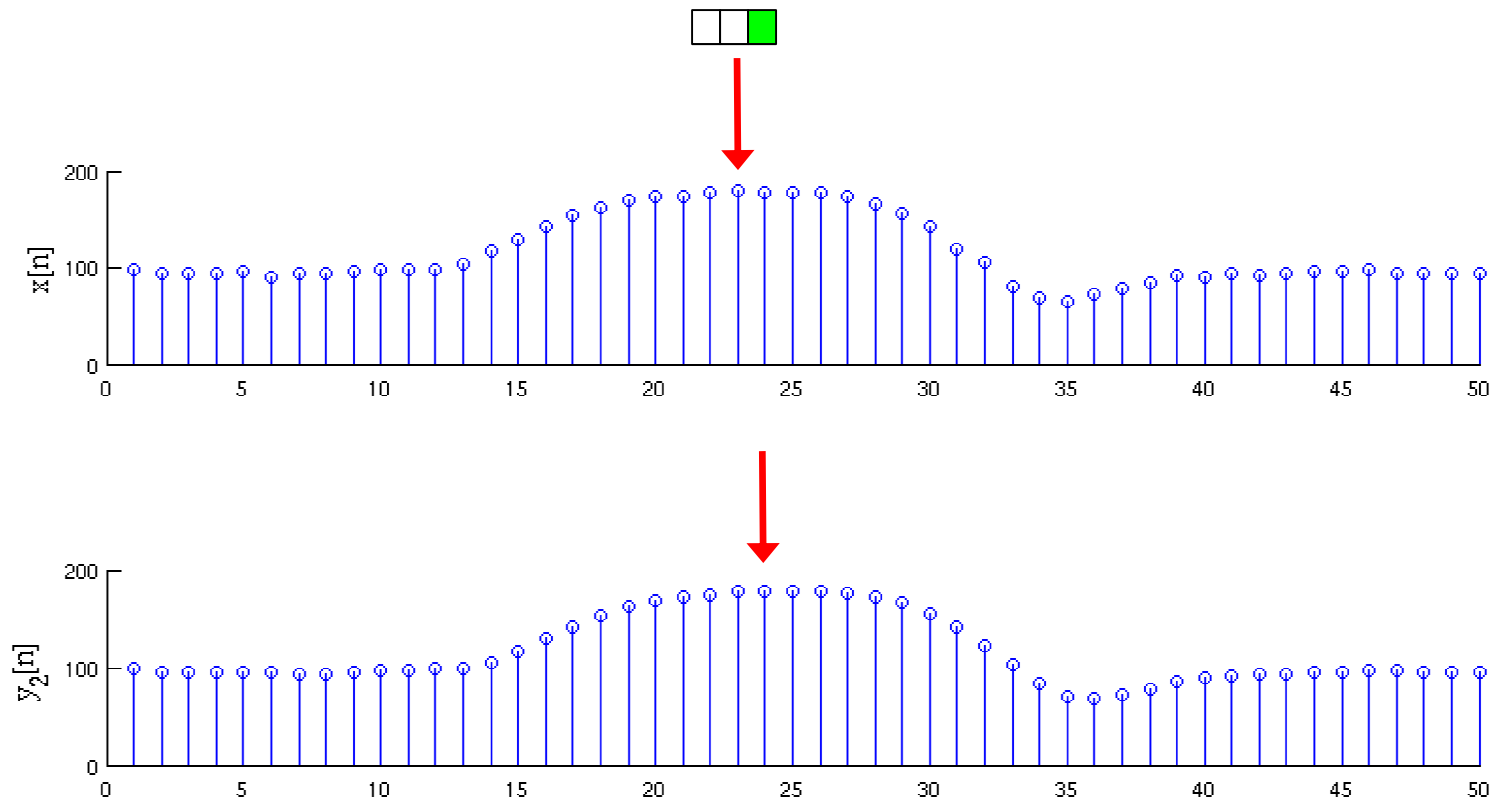
$$h_2[n] = h_1[n-1] = \frac{1}{4}\delta[n] + \frac{1}{2}\delta[n-1] + \frac{1}{4}\delta[n-2].$$

- Notice that  $H_2$  is causal, because  $h_2[n] = 0 \forall n < 0$ .
- The output of  $H_2$  is given by

$$\begin{aligned} y_2[n] &= x[n] * h_2[n] \\ &= \sum_{k=-\infty}^{\infty} h_2[k] x[n-k] \\ &= \frac{1}{4}x[n] + \frac{1}{2}x[n-1] + \frac{1}{4}x[n-2] \end{aligned}$$

- The output signal is still a weighted average of the input signal, but it is no longer centered.
- The output signal is *shifted* to the right by one sample relative to the input signal.
- In other words, the filter  $H_2$  introduces *delay* – a *phase shift*.

- “Sliding window” picture for  $H_2$ :



- The current output is a weighted average of the current input and the two previous inputs; the output signal is *delayed* or *shifted* by one sample relative to the input signal.

- Because  $h_2[n]$  does not have even symmetry, the frequency response  $H_2(e^{j\omega})$  takes complex values.
- It is given by

$$\begin{aligned}
 H_2(e^{j\omega}) &= \sum_{n=-\infty}^{\infty} h_2[n]e^{-j\omega n} \\
 &= \frac{1}{4} + \frac{1}{2}e^{-j\omega} + \frac{1}{4}e^{-j2\omega} \\
 &= \left[ \frac{1}{4}e^{j\omega} + \frac{1}{2} + \frac{1}{4}e^{-j\omega} \right] e^{-j\omega} \\
 &= H_1(e^{j\omega})e^{-j\omega}
 \end{aligned}$$

- The phase response is  $\arg H_2(e^{j\omega}) = -\omega$ .
- The filter  $H_2$  is causal and it could be implemented in real-time.
- But because the window is not “centered,” the output signal is shifted to the right by one sample relative to the input signal.

- In image processing, such shifting is usually *undesirable*.
- Usually, we do not want the output image to be shifted relative to the input image – we usually want them to be “lined up.”
- Moreover, in image processing we usually receive the whole image at once; i.e., all the pixels are in memory at the same time.
  - So there is no reason to restrict ourselves to causal processing!
  - Mathematically, for the output pixel  $J(m, n)$ , the input pixels  $I(m-1, n)$  and  $I(m, n-1)$  are *causal* or “past” inputs, while the input pixels  $I(m+1, n)$  and  $I(m, n+1)$  are *non-causal* or “future” inputs.
  - A filter that uses  $I(m+1, n)$  and  $I(m, n+1)$  to compute  $J(m, n)$  is *non-causal*.
  - But that’s usually not a problem in image processing – because we usually get all the pixels at the same time.
- For these reasons, in image processing the window (or structuring element) is usually symmetric, usually covers an odd number of pixels, and the reference point is usually in the center.
- This ensures that the output image is not shifted relative to the input image. If the filter is linear, then it has a zero phase response.

# More on Window Notation

- A **window**  $\mathbf{B}$  can be thought of as the set of pixel coordinates (or “offsets”)  $B_i = (p_i, q_i)$  of the covered pixels relative to the reference pixel  $(0, 0)$ :

$$\mathbf{B} = \{B_1, \dots, B_{2P+1}\} = \{(p_1, q_1), \dots, (p_{2P+1}, q_{2P+1})\}$$

## **Windows with a 1-D structure:**

$$\mathbf{B} = \text{ROW}(2P+1) = \{(0, -P), \dots, (0, P)\}$$

$$\mathbf{B} = \text{COL}(2P+1) = \{(-P, 0), \dots, (P, 0)\}$$

For example,  $\mathbf{B} = \text{ROW}(3) = \{(0, -1), (0, 0), (0, 1)\}$

## Window with a 2-D Structure

$$\mathbf{B} = \text{SQUARE}(9) = \{(-1, -1), (-1, 0), (-1, 1), \\ (0, -1), (0, 0), (0, 1), \\ (1, -1), (1, 0), (1, 1)\}$$

$$\mathbf{B} = \text{CROSS}(2P+1) = \text{ROW}(2P+1) \cup \text{COL}(2P+1)$$

$$\text{For example, } \mathbf{B} = \text{CROSS}(5) = \{ \quad \quad \quad (-1, 0), \\ \quad \quad \quad (0, -1), (0, 0), (0, 1), \\ \quad \quad \quad (1, 0) \quad \quad \quad \}$$

# The Windowed Set

- Given an image  $\mathbf{I}$  and a window  $\mathbf{B}$ , define the **windowed set** at  $(m, n)$  by:

$$\mathbf{B} \circ \mathbf{I}(m, n) = \{I(m+p, n+q); (p, q) \in \mathbf{B}\}$$

- It is the set of input pixels from the image that are **covered by  $\mathbf{B}$**  when the reference point is located at pixel  $(m, n)$ .
- This convolution-like formal definition of a simple concept will enable us to write simple and flexible definitions for binary filters.



- **B** = ROW(3):

$$\mathbf{B} \circ \mathbf{I}(m, n) = \{I(m, n-1), I(m, n), I(m, n+1)\}$$

- **B** = COL(3):

$$\mathbf{B} \circ \mathbf{I}(m, n) = \{I(m-1, n), I(m, n), I(m+1, n)\}$$

- **B** = SQUARE(9):

$$\begin{aligned} \mathbf{B} \circ \mathbf{I}(m, n) = \{ & I(m-1, n-1), I(m-1, n), I(m-1, n+1), \\ & I(m, n-1), I(m, n), I(m, n+1), \\ & I(m+1, n-1), I(m+1, n), I(m+1, n+1)\} \end{aligned}$$

- **B** = CROSS(5):

$$\mathbf{B} \circ \mathbf{I}(m, n) = \{ \begin{array}{l} I(m-1, n), \\ I(m, n-1), I(m, n), I(m, n+1), \\ I(m+1, n) \end{array} \}_{65}$$

# General Binary Filter

- Denote a logical operation  $G$  on the **windowed set**  $\mathbf{B} \circ \mathbf{I}(m, n)$  by

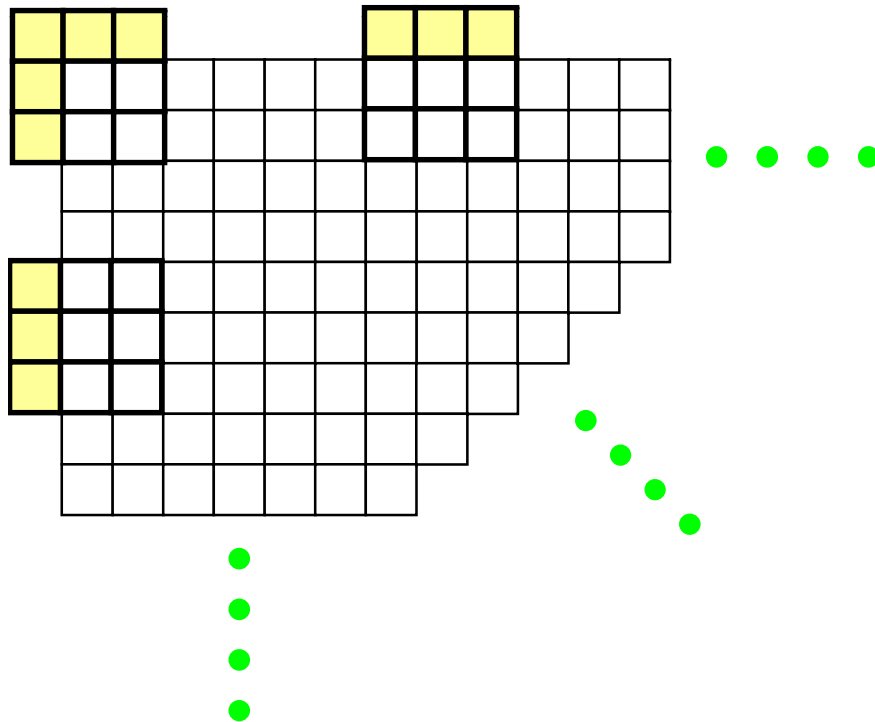
$$J(m, n) = G\{\mathbf{B} \circ \mathbf{I}(m, n)\} = G\{I(m+p, n+q); (p, q) \in \mathbf{B}\}$$

- By performing this at every pixel in the image, we obtain the **filtered image**

$$\mathbf{J} = G[\mathbf{I}, \mathbf{B}] = [J(m, n); 0 \leq m \leq M-1, 0 \leq n \leq N-1]$$

# Edge Effects

- What happens near the edges of the image when the window "hangs over" ?



- For now, we will fill the “empty space” by making extra copies of the image pixels on the outside rows & columns
- This is called “handling the edge effects by replication”

# Handling Edge Effects

- The best way to handle edge effects often depends on the particular application.
- **Unless otherwise specified**, for this class you should handle the edge effects as follows:
  - ▶ for morphological filtering, order statistic filtering, and other types of nonlinear filtering, handle the edge effects by replication.
  - ▶ for linear convolution in the image domain, handle the edge effects by zero padding.
  - ▶ for circular convolution involving the DFT/FFT, handle the edge effects by periodically extending the image.
  - ▶ for filtering processes involving the DCT, handle the edge effects by reflecting the image.
- These will be our **default** conventions in this class; the last three will make more sense later!

# Dilation Filter

- For a window **B** and a binary image **I**, we define the morphological filter

$$\mathbf{J} = \mathbf{DILATE}(\mathbf{I}, \mathbf{B})$$

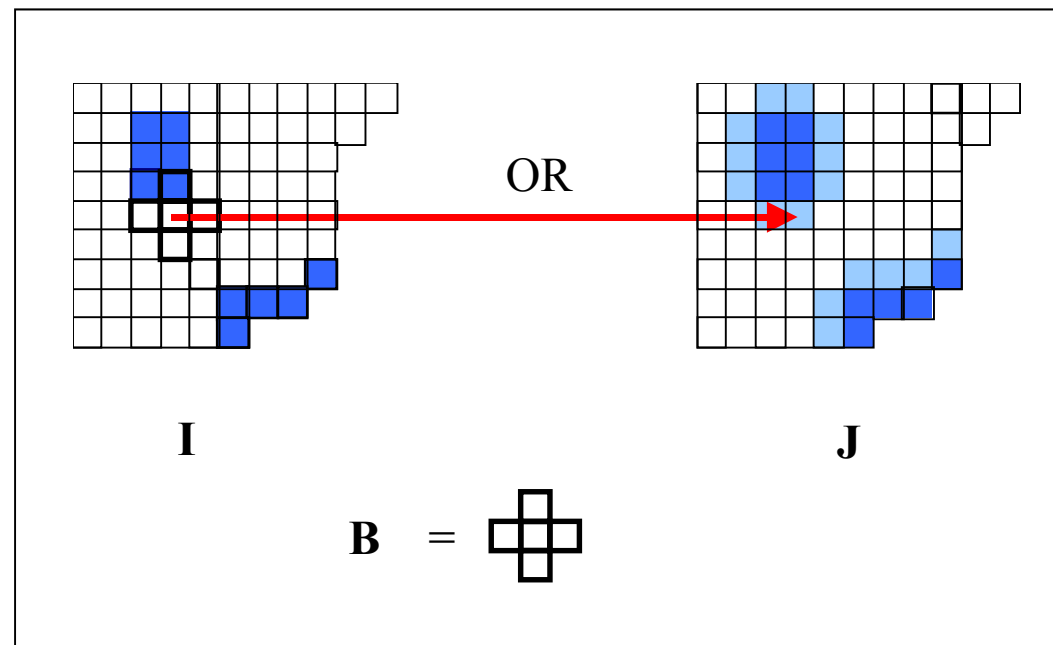
by

$$\begin{aligned} J(m, n) &= \text{OR} \{ \mathbf{B} \circ \mathbf{I}(m, n) \} \\ &= \text{OR} \{ \mathbf{I}(m+p, n+q); (p, q) \in \mathbf{B} \} \end{aligned}$$

- Often written as  $\mathbf{J} = \mathbf{I} \oplus \mathbf{B}$

# Dilation Filter

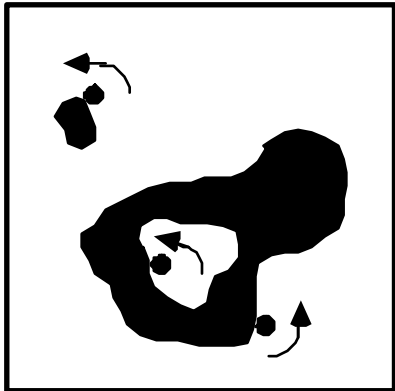
- So-called because DILATION **increases** the size of logical '1' objects.



Example of a local DILATION computation.

# Dilation Filter

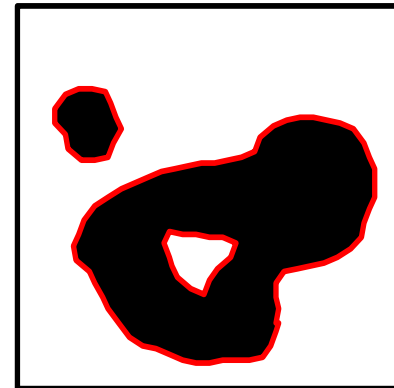
- Global effect of DILATION:



It is useful to think of the structuring element as rolling along all of the boundaries of all BLACK objects in the image.



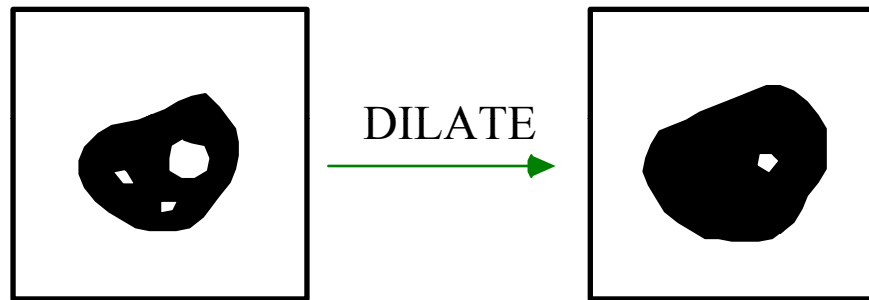
The center point of the structuring element traces out a set of paths.



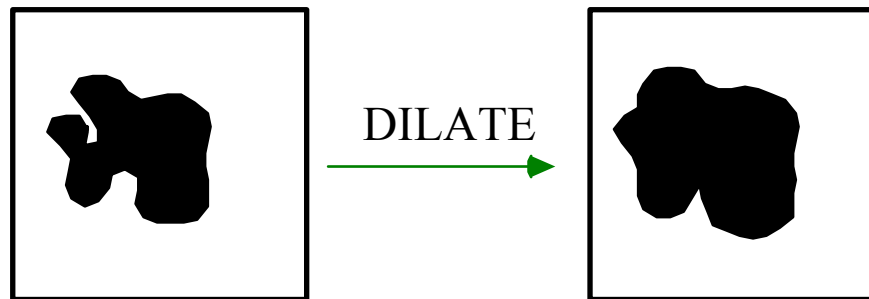
That form the boundaries of the dilated image.

# Qualitative Properties of Dilation

- Dilation **removes** holes that are **too small**:



- Dilation also **removes** gaps or bays that are **too narrow**:





# Erosion Filter

- For a window **B** and a binary image **I**, we define the morphological filter

$$\mathbf{J} = \mathbf{ERODE}(\mathbf{I}, \mathbf{B})$$

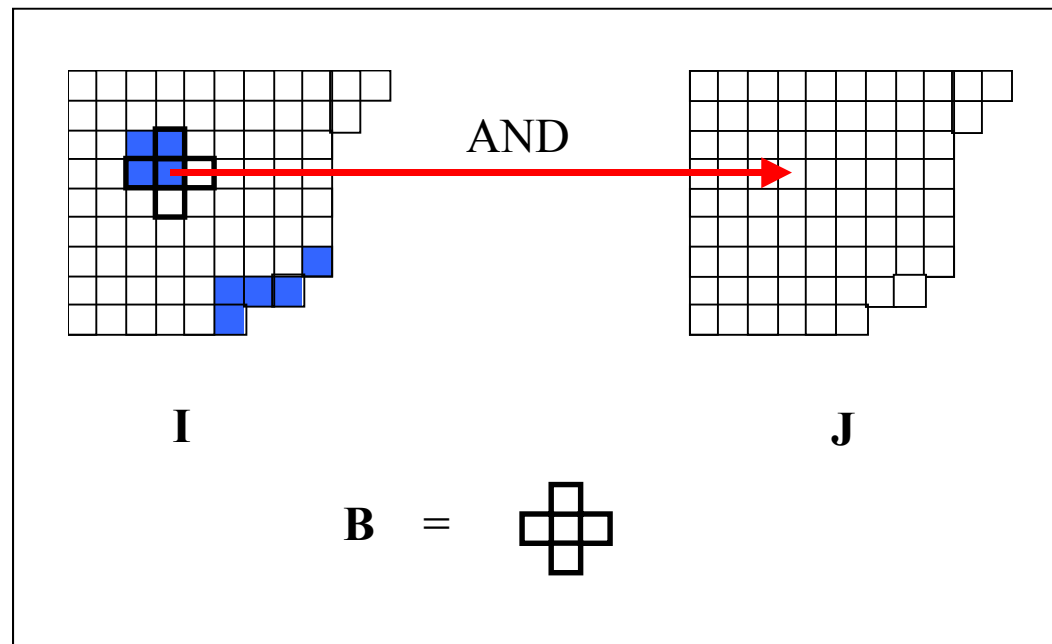
by

$$\begin{aligned} J(m, n) &= \text{AND} \{ \mathbf{B} \circ \mathbf{I}(m, n) \} \\ &= \text{AND} \{ \mathbf{I}(m+p, n+q); (p, q) \in \mathbf{B} \} \end{aligned}$$

- Often written as  $\mathbf{J} = \mathbf{I} \ominus \mathbf{B}$

# Erosion Filter

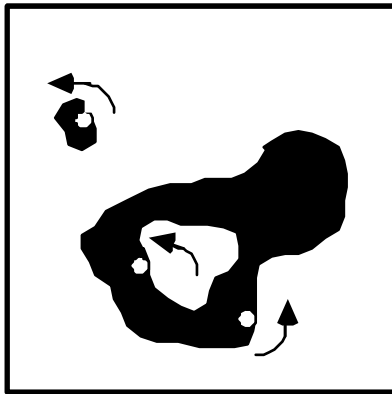
- So-called because **EROSION decreases** the size of logical '1' objects.



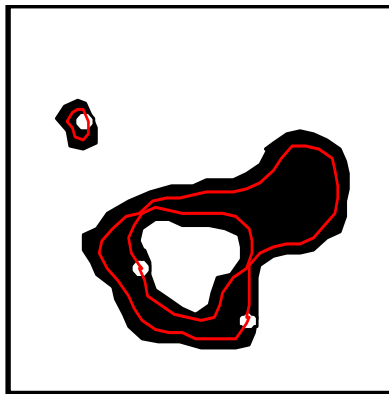
Example of a local  
EROSION  
computation.

# Erosion Filter

- Global effect of EROSION:



It is useful to think of the structuring element as rolling inside of the boundaries of all BLACK objects in the image.



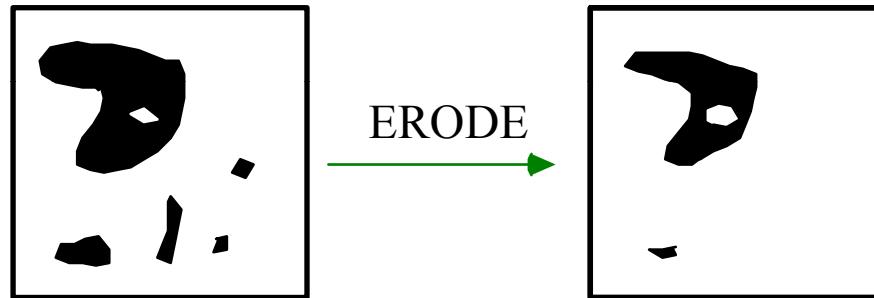
The center point of the structuring element traces out a set of paths.



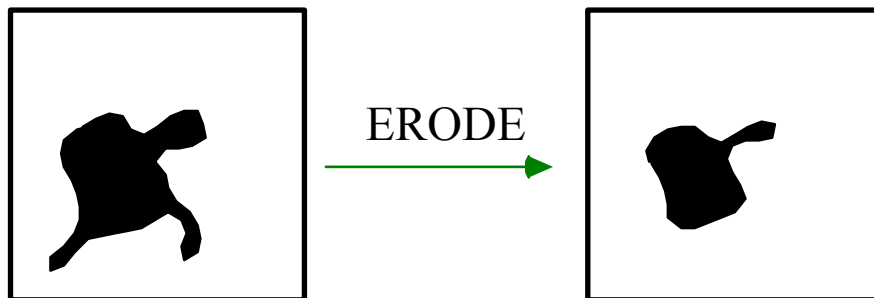
That form the boundaries of the eroded image.

# Qualitative Properties of Erosion

- Erosion removes objects that are **too small**:

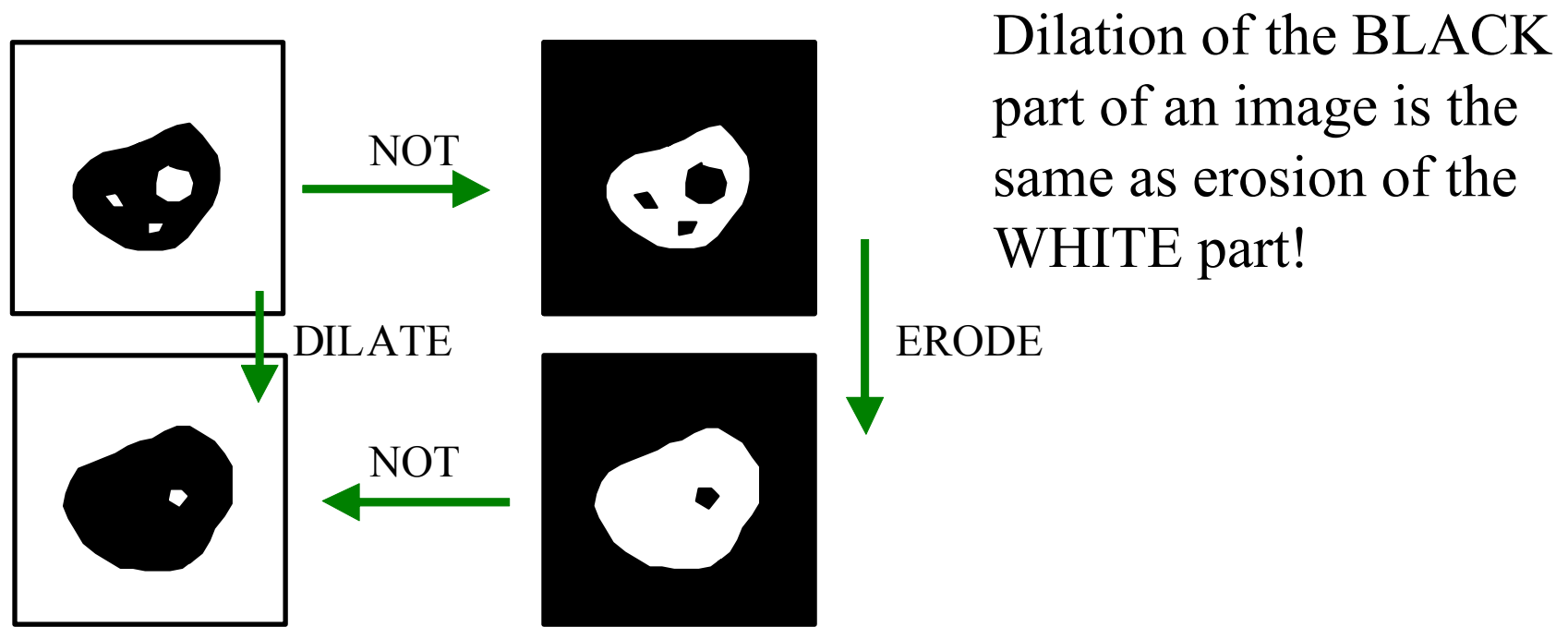


- Erosion also **removes** peninsulas that are **too narrow**:

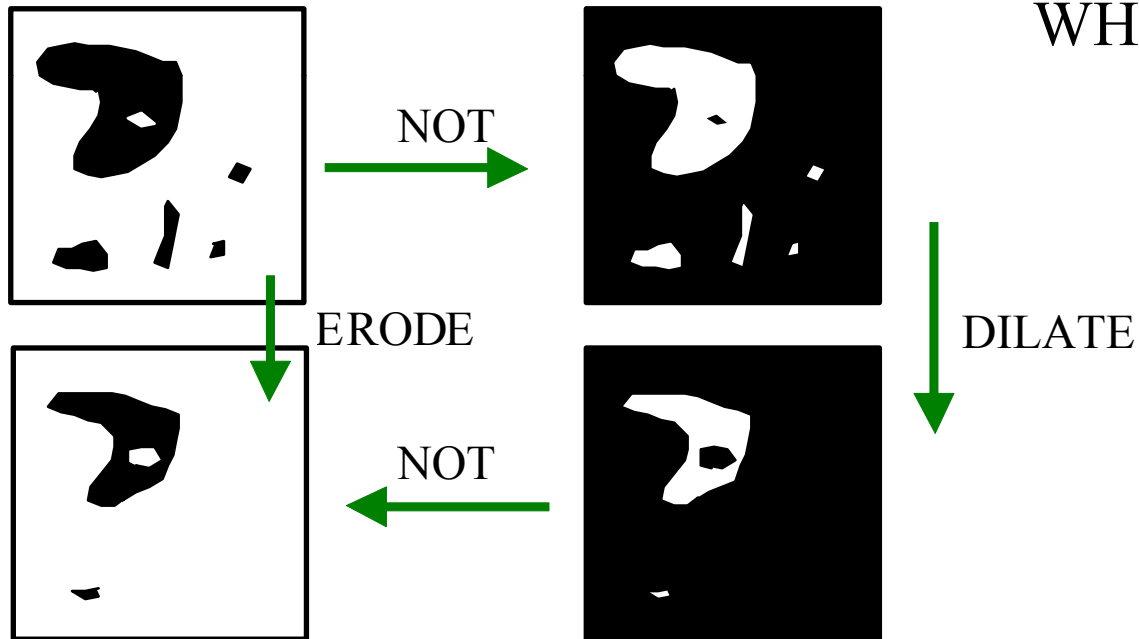


# Relating Erosion and Dilation

- Erosion and dilation are **dual** operations with respect to **complementation**.



Erosion of the BLACK part of an image is the same as dilation of the WHITE part!



# Relating Erosion and Dilation

- $\text{NOT}(\mathbf{I} \oplus \mathbf{B}) = \text{NOT}(\mathbf{I}) \ominus \mathbf{B}$  (dual operations)
- However, erosion and dilation are only **approximate** inverses of one another.
- Dilation enlarges the objects.
  - An erosion will return the objects to their original sizes
  - but it cannot restore holes/gaps that were filled in
- Erosion shrinks the objects
  - A dilation will return the objects to their original sizes
  - but it cannot restore the objects/peninsulas that were removed by erosion

# Median (Majority) Filter

- For a window **B** and a binary image **I**, we define the morphological filter

$$\mathbf{J} = \mathbf{MEDIAN}(\mathbf{I}, \mathbf{B})$$

or

$$\mathbf{J} = \mathbf{MAJORITY}(\mathbf{I}, \mathbf{B})$$

by

$$\begin{aligned} J(m, n) &= \text{MAJ} \{ \mathbf{B} \circ \mathbf{I}(m, n) \} \\ &= \text{MAJ} \{ \mathbf{I}(m+p, n+q); (p, q) \in \mathbf{B} \} \end{aligned}$$

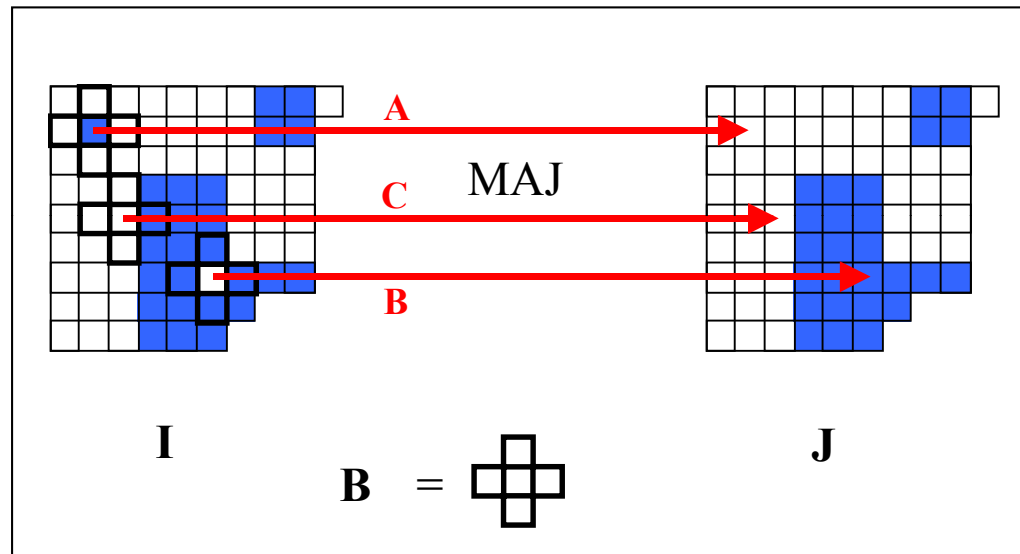


# Median Filter

- Removes both objects and holes of insufficient size, as well as both gaps (bays) and peninsulas of insufficient width.
- But generally does not change the overall size of objects (although it does alter them).
- Binary median is self-dual with respect to complementation:

$$\text{MEDIAN}[ \text{NOT}(\mathbf{I}) ] = \text{NOT}[ \text{MEDIAN}(\mathbf{I}) ]$$

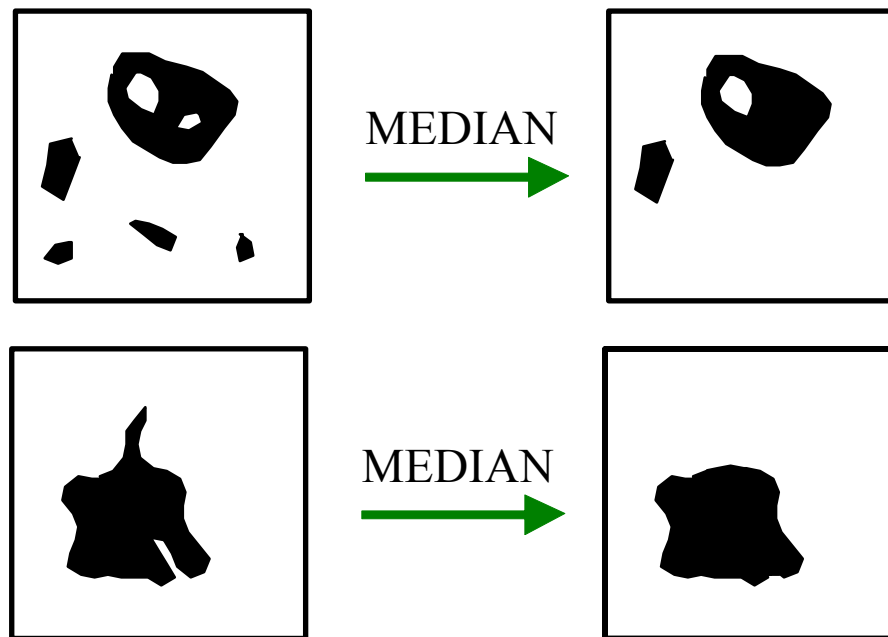
# Median Filter



The median removed the small **object A** and the small hole **hole B** but did not change the boundary (**size**) of the larger region **C**

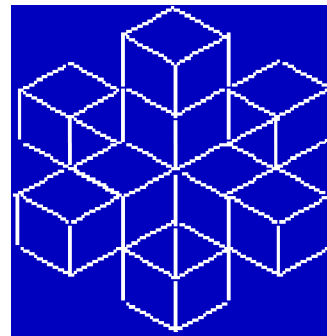
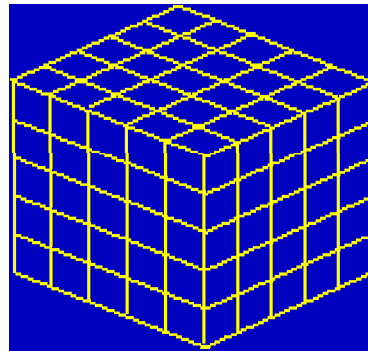
# Qualitative Properties of Median

- Median removes both **objects** and **holes** that are **too small**, as well as both **gaps (bays)** and **peninsulas** that are **too narrow**:



# 3-D Median Filter Example

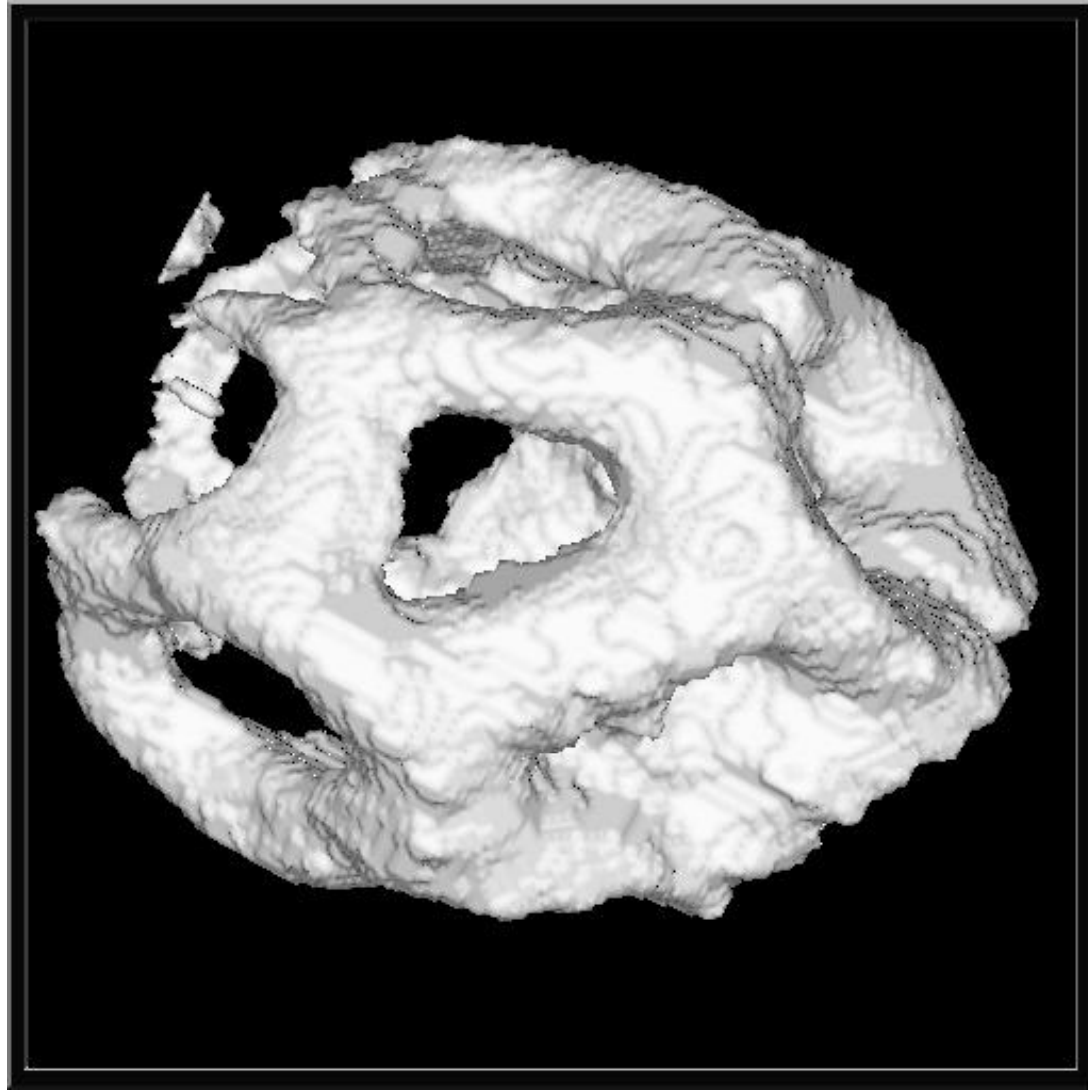
- The following example is a 3-D Laser Scanning Confocal Microscope (LSCM) image (binarized) of a pollen grain. Magnification » 200x



Examples of 3-D windows: CUBE(125) and CROSS<sub>3-D</sub>(13)



**LSCM image of pollen grain**



**Pollen grain image filtered with CUBE(125) binary  
median filter**

# OPEN and CLOSE

- Define **new** morphological operations by performing the basic ones in sequence.
- Given an image **I** and window **B**, define

$$\text{OPEN}(\mathbf{I}, \mathbf{B}) = \text{DILATE} [\text{ERODE}(\mathbf{I}, \mathbf{B}), \mathbf{B}]$$

$$\text{or } \mathbf{I} \circ \mathbf{B} = (\mathbf{I} \ominus \mathbf{B}) \oplus \mathbf{B}$$

$$\text{CLOSE}(\mathbf{I}, \mathbf{B}) = \text{ERODE} [\text{DILATE}(\mathbf{I}, \mathbf{B}), \mathbf{B}]$$

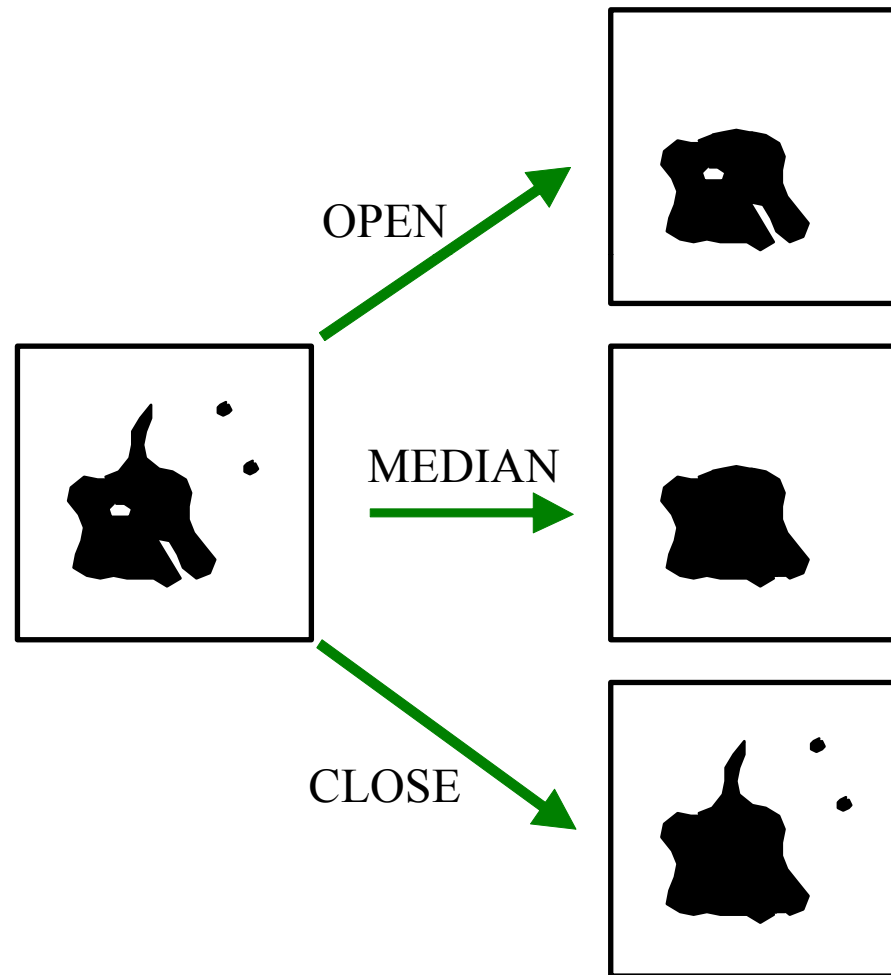
$$\text{or } \mathbf{I} \bullet \mathbf{B} = (\mathbf{I} \oplus \mathbf{B}) \ominus \mathbf{B}$$

# OPEN and CLOSE

- In other words,
  - OPEN = erosion (by **B**) followed by dilation (by **B**)
  - CLOSE = dilation (by **B**) followed by erosion (by **B**)
- Compared to MEDIAN, which removes **both** objects and holes that are too small, OPEN and CLOSE are “asymetric” smoothers that remove **either** objects **or** holes:
  - OPEN removes small objects/fingers (better than MEDIAN), but not holes, gaps, or bays.
  - CLOSE removes small holes/gaps (better than MEDIAN) but not objects or peninsulas.
- OPEN and CLOSE generally **do not affect object size**.
- They are specialized **biased smoothers**.



# OPEN and CLOSE vs. Median



## **OPEN-CLOSE and CLOSE-OPEN**

- Very effective smoothers can be obtained by sequencing the OPEN and CLOSE operators:

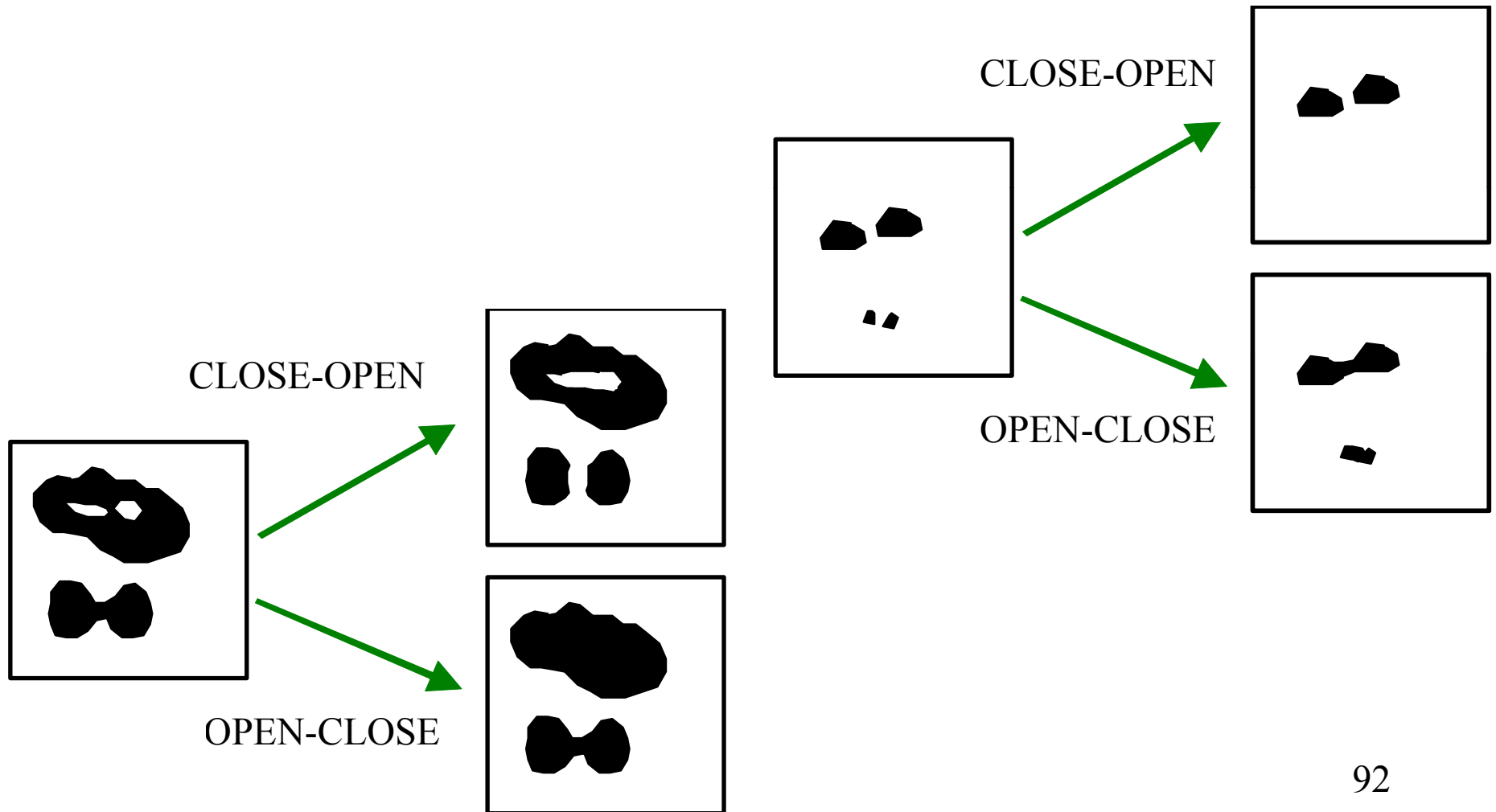
$\text{OPEN-CLOSE}(\mathbf{I}, \mathbf{B}) = \text{OPEN} [\text{CLOSE} (\mathbf{I}, \mathbf{B}), \mathbf{B}]$

$\text{CLOSE-OPEN}(\mathbf{I}, \mathbf{B}) = \text{CLOSE} [\text{OPEN} (\mathbf{I}, \mathbf{B}), \mathbf{B}]$

## OPEN-CLOSE and CLOSE-OPEN

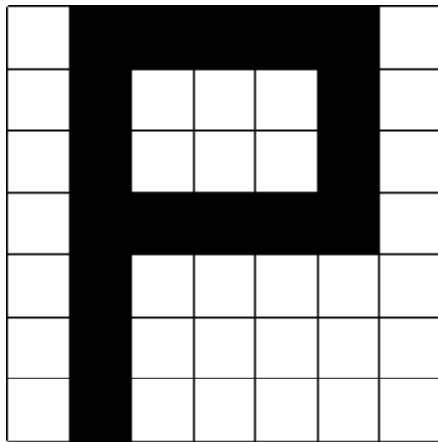
- These operations are similar (but not identical).
- Both remove small structures w/o affecting size.
- Both similar to the median filter but smooth **more** (for a given structuring element **B**).
- Notable differences between OPEN-CLOSE and CLOSE-OPEN:
  - OPEN-CLOSE tends to **link neighboring objects together**
  - CLOSE-OPEN tends to **link neighboring holes together**

# Comparing OPEN-CLOSE and CLOSE-OPEN



# Binary Template Matching

- Often, it is of interest to **find** instances of a certain **object** in a binary image.
- This can be accomplished with simple logical operations and a **template**.
- The template is similar to a structuring element, but the pixels of the template have logical values ('1' or '0').



a template for the letter “P”

the template is like a “mini image”

# Binary Match Function

- For two binary variables A and B, define MATCH(A,B) by

| A | B | MATCH(A,B) |
|---|---|------------|
| 0 | 0 | 1          |
| 0 | 1 | 0          |
| 1 | 0 | 0          |
| 1 | 1 | 1          |

- NOTE: this is just the binary XNOR function.
- Define** the MATCH between two images **I** and **J** by  
 $\mathbf{K} = \text{MATCH}(\mathbf{I}, \mathbf{J})$  if  $K(m,n) = \text{MATCH}[I(m,n), J(m,n)] \forall (m,n)$

# Binary Match Measures

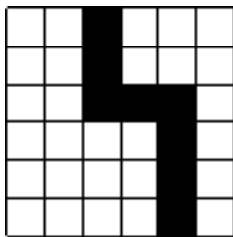
- Let  $\mathbf{K} = \text{MATCH}(\mathbf{I}, \mathbf{J})$ .
- Define:

$$M_1(\mathbf{I}, \mathbf{J}) = \text{AND}[ K(m,n) \forall (m,n) ]$$

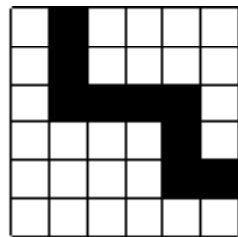
(reveals perfect matches only)

$$M_2(\mathbf{I}, \mathbf{J}) = \text{SUM}[ K(m,n) \forall (m,n) ]$$

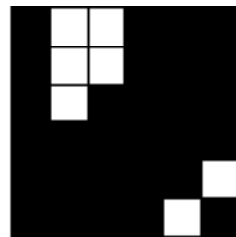
(measures how good the match is)



**I**



**J**



**K=MATCH(I, J)**

$$M_1(\mathbf{I}, \mathbf{J}) = '0'$$

$$M_2(\mathbf{I}, \mathbf{J}) = 29$$

# Template Matching

- When the template  $\mathbf{T}$  is centered at pixel  $(m,n)$ , we define

$$\text{MATCH}(\mathbf{I}, \mathbf{T}; m, n)$$

to be the MATCH of  $\mathbf{T}$  and that part of  $\mathbf{I}$  that is covered by  $\mathbf{T}$ . In other words, between  $\mathbf{T}$  and a “mini-image” from  $\mathbf{I}$  that is the same size and shape as  $\mathbf{T}$ .

- The binary template matching operation can then apply either match measure  $M_1$  or  $M_2$  to this “mini-image.”
- By doing this at every pixel  $(m, n)$ , we obtain a full-size match measure image  $M_1(\mathbf{I}, \mathbf{T}; m, n)$  or  $M_2(\mathbf{I}, \mathbf{T}; m, n)$ .
- $M_1(\mathbf{I}, \mathbf{T}; m, n)$  is ‘1’ where the neighborhood about pixel  $(m,n)$  in  $\mathbf{I}$  matches the template **exactly** and is ‘0’ elsewhere.
- Each pixel of  $M_2(\mathbf{I}, \mathbf{T}; m, n)$  is an integer that gives the count of how many pixels in the neighborhood about  $(m,n)$  match the template. This count quantifies how well the neighborhood “matches” the template.



# Target Detection by Matched Filtering

- Because it is a filtering operation that is based on matching, template matching is also called **matched filtering**.
- It is used often in target detection applications.
- The **goal** is to find instances of the template **T** in the image **I**; i.e. to produce a binary “map” image that is ‘1’ at pixels where **I** contains an instance of **T** and is ‘0’ elsewhere.
- If we are only interested in detecting **exact** matches, then the binary image  $M_1(\mathbf{I}, \mathbf{T}; m, n)$  can be taken as the final result.
- Except in controlled environments, this is usually impractical because of noise and variations in the actual target appearance relative to the model **T** (such as rotation, magnification, partial occlusion, etc).
- Thus, we often threshold the integer image  $M_2(\mathbf{I}, \mathbf{T}; m, n)$  instead to obtain a binary “map” image that is ‘1’ where the match is “close enough” to declare a detection.

# Target Detection by Matched Filtering

- As usual, choosing the **best** value for the threshold can be **difficult**.
- If we have *a priori* knowledge that there is **only one** instance of the target in **I**, then it suffices to set the binary “map” image to ‘1’ at the pixel where  $M_2(\mathbf{I}, \mathbf{T}; m, n)$  is maximized (and to ‘0’ elsewhere).
- In applications where there may be multiple instances of the target in **I**, one common approach is to convert the integer match image  $M_2(\mathbf{I}, \mathbf{T}; m, n)$  to a floating point “percentage match” image.
  - This is done by dividing  $M_2(\mathbf{I}, \mathbf{T}; m, n)$  by the number of pixels in **T**.
  - The “percentage match” image can then be thresholded with a threshold that is between zero and one.
- There are two advantages to this approach:
  1. It tends to make the task of choosing the threshold conceptually easier; for example: “declare a detection if the match is 95% or better.”
  2. It saves us from having to continually adjust the threshold in applications where the size of the target (and the template) may be changing over time – as in a video sequence where the target is moving.

# Binary Image Compression

- In later Modules we will discuss gray-level and color image and video compression at length.
- It is also of interest to compress binary images.
- The idea is to **exploit redundancy**.

# Run-length Coding

- The number of bits required to store an  $M \times N$  binary image is  $MN$ .
- Usually, this can be significantly reduced by **coding** the image.
  - This kind of coding is called **source coding**; the goal is to achieve compression.
  - It is different from **channel coding**, where the goal is to **add** bits in order to achieve error resiliency.
- Run-length coding works well if the image contains long runs of '1' pixels and/or '0' pixels.

# How Run-Length Coding Works

- **Algorithm:** For image row  $m$ :
  - (1) Store the first pixel value ('0' or '1') in row  $m$  as a reference
  - (2) Set **run counter**  $c = 1$
  - (3) For each pixel in the row:
    - Examine the next pixel to the right
    - If same as current pixel, set  $c = c + 1$
    - If different from current pixel, **store**  $c$  and set  $c = 1$
    - Continue until end of row is reached
- Each run-length is stored using  **$b$  bits**.

# Run-Length Code

## Example


what's  
stored: '1'    7            5            8            3    1  
row m  • • • •

## Comments on Run-Length Coding

- Excellent compression on some images.
- If the image contains many runs of 1's and 0's.
- Baseline JPEG uses it to code the DCT coefficients.

# Run-Length Code

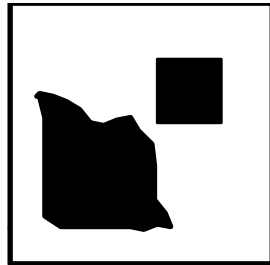
- If the image contains only short runs, run-length coding can **increase** the required storage.

what's  
stored: '1'1  
row m 

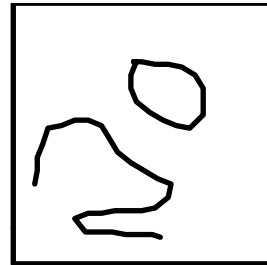
- Each count is stored with  $b$  bits!
- Worst-case example: storage increased  **$b$ -fold**!
- Rule of thumb: average run-length  $L > b$ .

# Contour Representation

- Two types of binary images: **region images** and **contour images**.



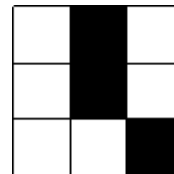
region image



contour image

- Contour images consist only of single-pixel width contours (straight and/or curved) and single points.
- Each '1' pixel in a contour image can have **at most** two 8-neighbors that are also '1'.

Example of a neighborhood  
from a contour image

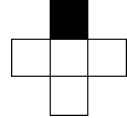




# Approximate Contour Generation

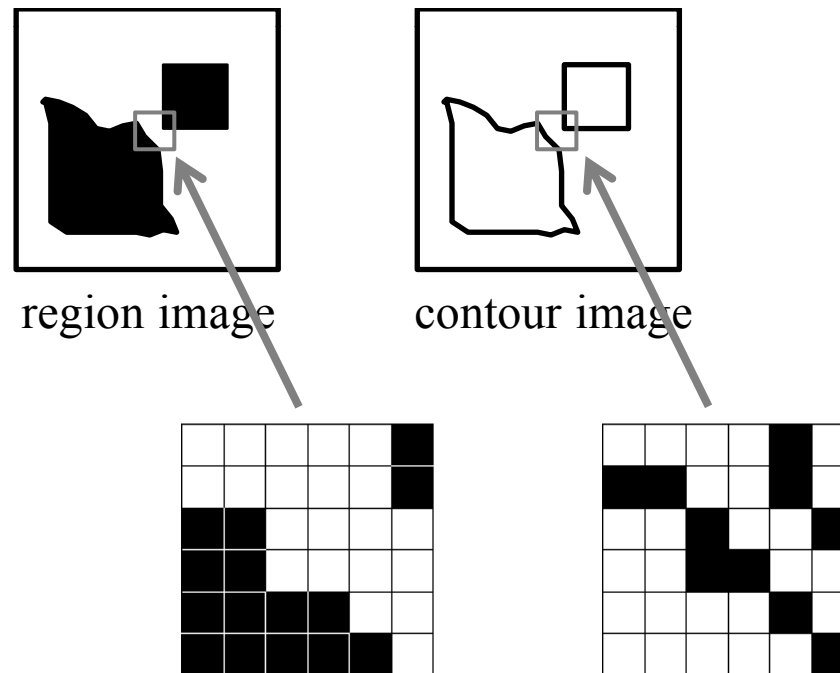
- A binary region image can be converted into an **approximate** contour image by the following procedure.
- Let **I** be a region image. Define a new binary image **J** according to:

$J(m, n) = '1'$  if  $I(m, n) = '0'$  and  $I(m, n)$  has one or more 4-neighbors that are  $'1'$

- A  $'0'$  pixel and it's 4-neighbors:  (one of them is  $'1'$ )
- Intuitively, this algorithm traces out a contour of  $'1'$  pixels around any region (object) of  $'1'$  pixels in the input image.
- A  $'1'$  pixel is output any time the window is on a  $'0'$  pixel in the input image but touches a  $'1'$  pixel.

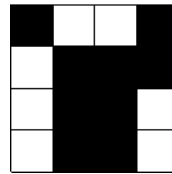
# Approximate Contour Generation

- Example:



# Approximate Contour Generation

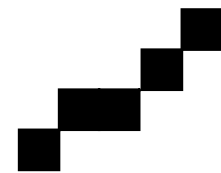
- This algorithm is simple and easy to implement, but it is **approximate**.
- The output image is **not guaranteed** to be a true contour image.
- There may be some '1' pixels in the output image that have more than two 8-neighbors which are also '1'.
- Example:



- When this occurs, post processing such as **edge thinning** must generally be applied to make corrections.
- For this course, we won't be too concerned about it.
- Later, we will study more complicated **edge detection** algorithms that **can** be guaranteed to produce a true contour image as output.

# Chain Coding

- The chain code is a highly efficient method for **coding binary contour images**.
- If the **initial** coordinate of an **8-connected contour** is specified, then the rest of the contour can be represented by storing a sequence of direction codes for how to traverse the contour.



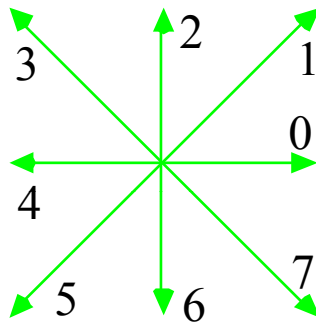
contour



initial point  
and directions

# Chain Code

- We use the following 8-neighbor **direction codes**:

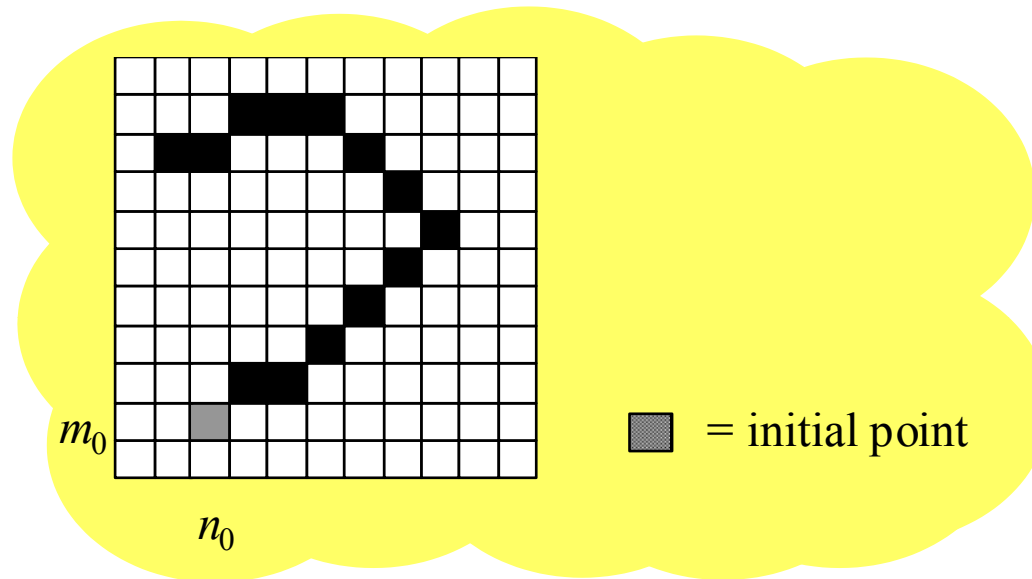


- Since the numbers 0, 1, 2, 3, 4, 5, 6, 7 can be coded by their **3-bit binary equivalents**:

000, 001, 010, 011, 100, 101, 110, 111

the location of each point on the contour **after** the initial point can be coded by **3 bits**.

# Chain Code Example



- Its chain code (after recording the initial coordinate  $(m_0, n_0)$ ):

1, 0, 1, 1, 1, 1, 3, 3, 3, 4, 4, 5, 4

= 001, 000, 001, 001, 001, 001, 011, 011, 011, 100, 100, 101, 100

# Chain Code Comments

- The end of the code is marked by a “flag”. One option — give the direction that “goes back” on itself: e.g., a ‘4’ and then a ‘0’.
- The **compression** can be quite significant: coding by  $B$ -bit coordinates ( $B = 9$  for 512 x 512 images) requires 6x as much storage.
- Chain codes are effective for many computer vision and pattern recognition applications, e.g. **character recognition**.
- **Comment:** for closed contours, the initial coordinate can be chosen arbitrarily. If the contour is **open**, then it should be an **end point**.

## Module 2 Comments

- Many things can be accomplished with binary images – since **shape** is often well-preserved when a grayscale image is thresholded.
- However, grayscale images are also important. We will deal with **gray scales** next, but **not shape** – onward to **Module 3**.