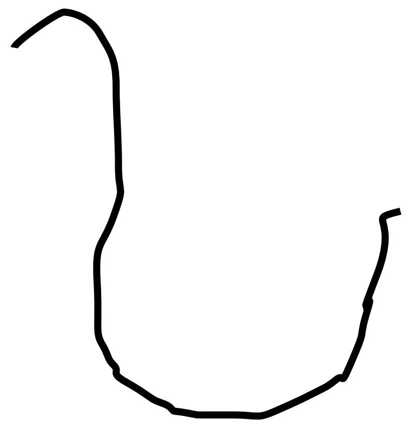
Building Recognizers for Digital Ink and Gestures

Digital Ink

- Natural medium for pen-based computing
  - Pen inputs strokes
  - Strokes recorded as lists of X,Y coordinates
  - E.g., in Java:
    - Point[] aStroke;

- Can be used as *data* -- handwritten content
- ... or as *commands* -- gestures to be processed

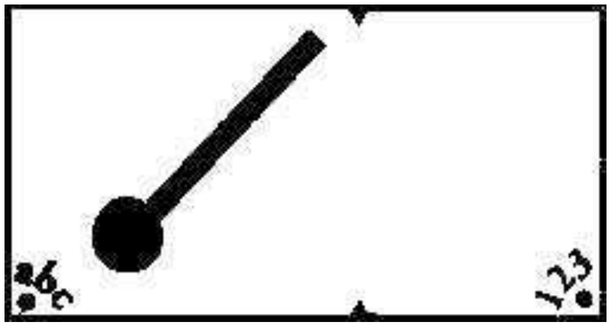


Distinguishing Content from Commands

- Depends on the set of input devices, but ....
  - generally **modal**
  - Meaning that you're either in content mode or you're in command mode
- Often a button or other model selector to indicate command mode
  - Example: Wacom tablet pen has a mode button on the barrel
  - Temporary switch--only changes mode while held down, rather than a toggle.

Other Options

- Use a special character that disambiguates from content input and command input
  - E.g., graffiti on PalmOS
  - "Command stroke" says that what comes after is meant to be interpreted as a command.



- Can also have special "alphabet" of symbols that are unique to commands
- Can also use another interactor (e.g., the keyboard)
  - but requires that you put down the pen to enter commands

Still More Options

- "Contextually aware" commands
- Interpretation of whether something is a command or not depends on where it is drawn
  - E.g., Igarashi's Pegasus drawing beautification program
    - a scribble in free space is content
    - a scribble that multi-crosses another line is interpreted as an erase gesture

“Sketch-based” user interfaces

- User interfaces aimed at creating, refining, and reusing hand-drawn input
- Typically:
  - Few “normal” GUI controls
  - Strokes contextually interpreted, and intermingled with content
- Examples:
  - Drawing beautification (Igarashi: Pegasus)
  - UI creation (Landay: SILK)
  - Turn UML, diagrams, etc., into machine representations (Saund)
  - 3D modeling (Igarashi: Teddy)

Why Use Ink as Commands?

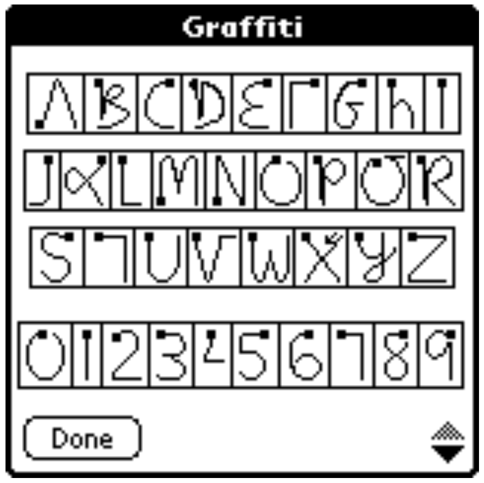
- Avoids having to use another interactor as the "command interactor"
  - Example: don't want to have to put down the pen and pick up the keyboard
- What's the challenge this with, though?
  - The command gestures *have* to be interpreted by the system
  - Needs to be reliable, or undoable/correctable
  - In contrast to content:
    - For some applications, uninterpreted content ink may be just fine

Content Recognizers

- Feature-based recognizers:
- Canonical example: Dean Rubine, *The Automatic Recognition of Gestures*, Ph.D. dissertation, CMU 1990.
  - "Feature based" recognizer, computes range of metrics such as length, distance between first and last points, cosine of initial angle, etc
  - Compute a *feature vector* that describes the stroke
  - Compare to feature vector derived from training data to determine match (multidimensional distance function)
  - To work well, requires that values of each feature should be normally distributed within a gesture, and between gestures the values of each feature should vary greatly

Content Recognizers [2]

- "Unistrokes" (a la PalmOS Graffiti)
- Use a custom alphabet with high-disambiguation potential
- Decompose entered strokes into constituent strokes and compare against template
    - E.g., unistrokes uses 5 different strokes written in four different orientations (0, 45, 90, and 135 degrees)
- Little customizability, but good recognition results and high data entry speed
- Canonical reference:
    - D. Goldberg and C. Richardson, *Touch-Typing with a Stylus*. Proceedings of CHI 1993.



Content Recognizers [3]

- Waaaaay more complex types of recognizers that are out of the scope of this class
  - E.g., neural net-based, etc.

This Lecture

- Focus on recognition techniques suitable for **command gestures**
- While we can build these using the same techniques used for **content ink**, we can also get away with some significantly easier methods
  - Read: "hacks"
- Building general-purpose recognizers suitable for large alphabets (such as arbitrary text) is outside the scope of this class
- We'll look at two simple recognizers:
  - 9-square
  - Siger

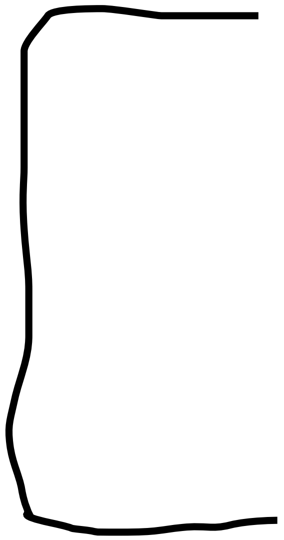
9-square

- Useful for recognizing "Tivoli-like" commands
- Developed at Xerox PARC for use on the Liveboard system
    - Liveboard [1992]: 4 foot X 3 foot display wall with pen input
- Used in "real life" meetings over a period of several years, supported digital ink and natural ink gestures

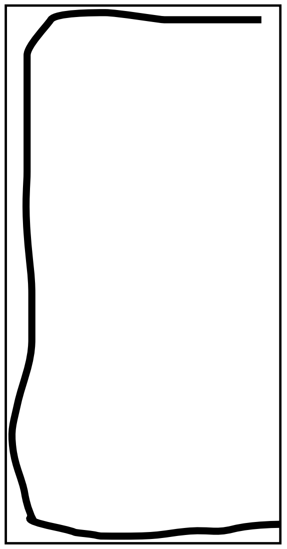
“9 Square” recognizer

- Basic version of algorithm:
  1. Take any stroke
  2. Compute its bounding box
  3. Divide the bounding box into a 9-square tic-tac-toe grid
  4. Mark which squares the stroke passes through
  5. Compare this with a template

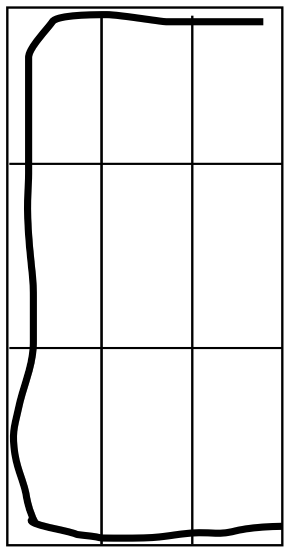
1. Original Stroke



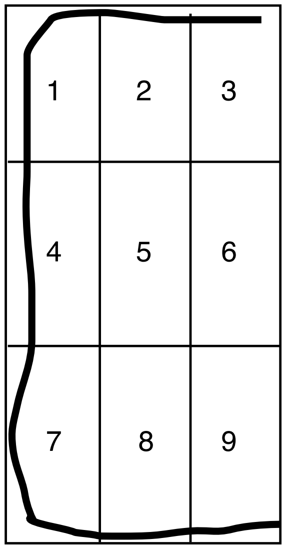
2. Compute Bounding Box



3. Divide Bounding Box into 9 Squares (3x3 grid)

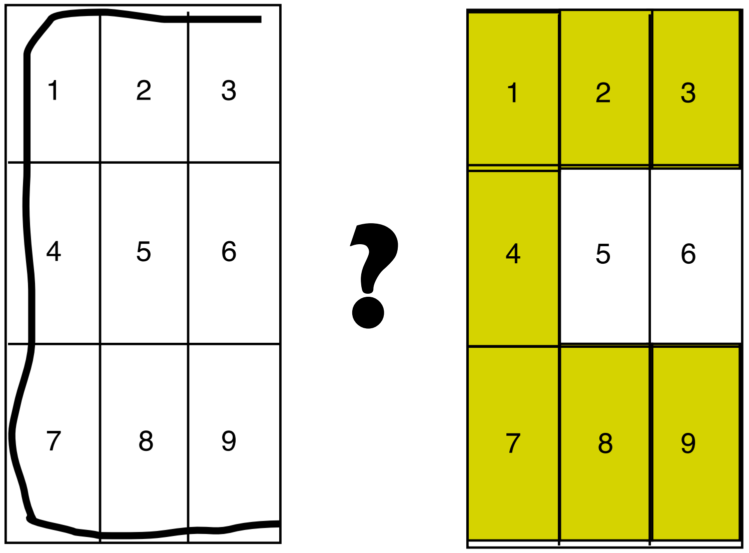


4. Mark Squares Through Which the Stroke Passes



representation: [X, X, X,
X, 0, 0,
X, X, X]

5. Compare with Template



stroke: [X, X, X,
X, 0, 0,
X, X, X]

=

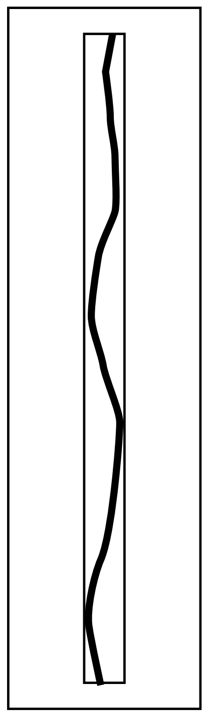
template: [X, X, X,
X, 0, 0,
X, X, X]

Implementing 9-square

- Create set of templates that represent the intersection squares for the gestures you want to recognize
- Bound the gesture, 9-square it, and create a vector of intersection squares
- Compare the vector with each template vector to see if a match occurs

Gotchas [1]

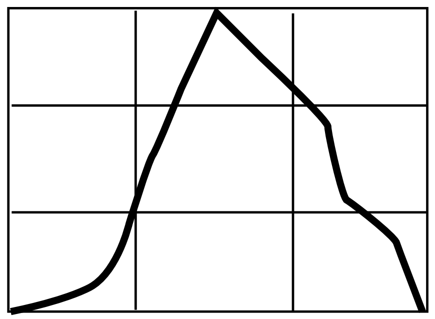
- What about long, narrow gestures (like a vertical line?)
- Unpredictable slicing
  - A perfectly straight vertical line has a width of 1, impossible to subdivide
  - More likely, a narrow but slightly uneven line will cross into and out of the left and right columns
- Solution: pad the bounding box before subdividing
  - Can just pad by a fixed amount, or
  - Pad separately in each dimension
    - Long vertical shapes may need more padding in the horizontal dimension
    - Long horizontal shapes may need more padding in the vertical dimension
    - Compute a pad factor for each dimension based on the other



Gotchas [2]

- Hard to do some useful shapes, e.g., vertical caret
- Is the correct template

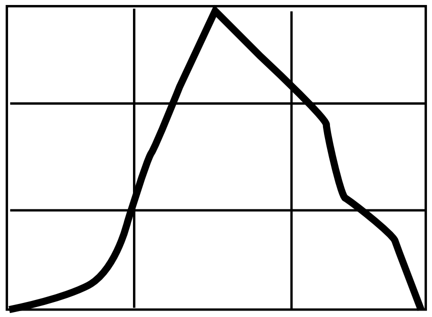
  [0, X, 0,
0, X, 0,
X, 0, X] or.... [0, X, 0,
X, 0, X,
X, 0, X]
- ... or other similar templates?
- Inherent ambiguity in matching the symbol as it is likely to be drawn to the 9-square template
- Any good solutions?



Gotchas [2]

- Hard to do some useful shapes, e.g., vertical caret
- Is the correct template

```
[0, X, 0,                [0, X, 0,
 0, X, 0,     or....      X, X, X,
 X, 0, X]                 X, 0, X]
```



- ... or other, similar templates?
- Inherent ambiguity in matching the symbol as it is likely to be drawn to the 9-square template
- Any good solutions?
- **Represent that ambiguity**
- Introduce a “don’t care” symbol into the template

Don’t Cares

- Use 0 to represent no intersection
- Use X to represent intersection
- Use * to represent don’t cares

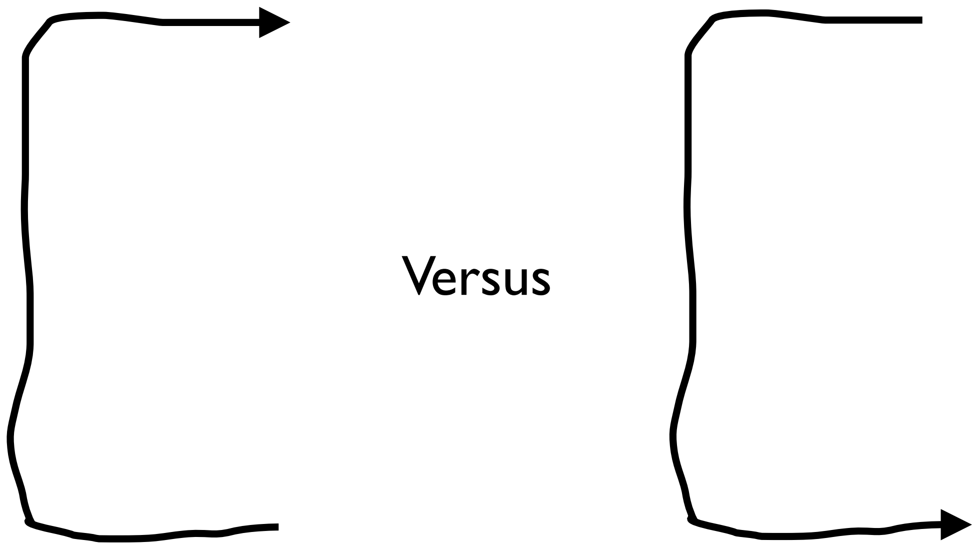
- Example:

```
[0, X, 0,            [0, X, 0,
 *, *, *,      or...  *, X, *,
 X, 0, X]             X, 0, X]
```

- Now need custom matching process (simple equivalence testing is not “smart enough”)
- if stroke[i] == template[i] || template[i] == “*”

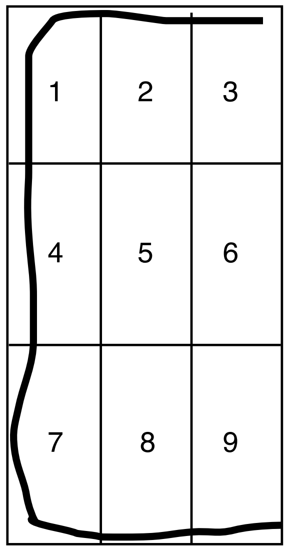
An Enhancement

- What if we want direction to matter?
- Example:

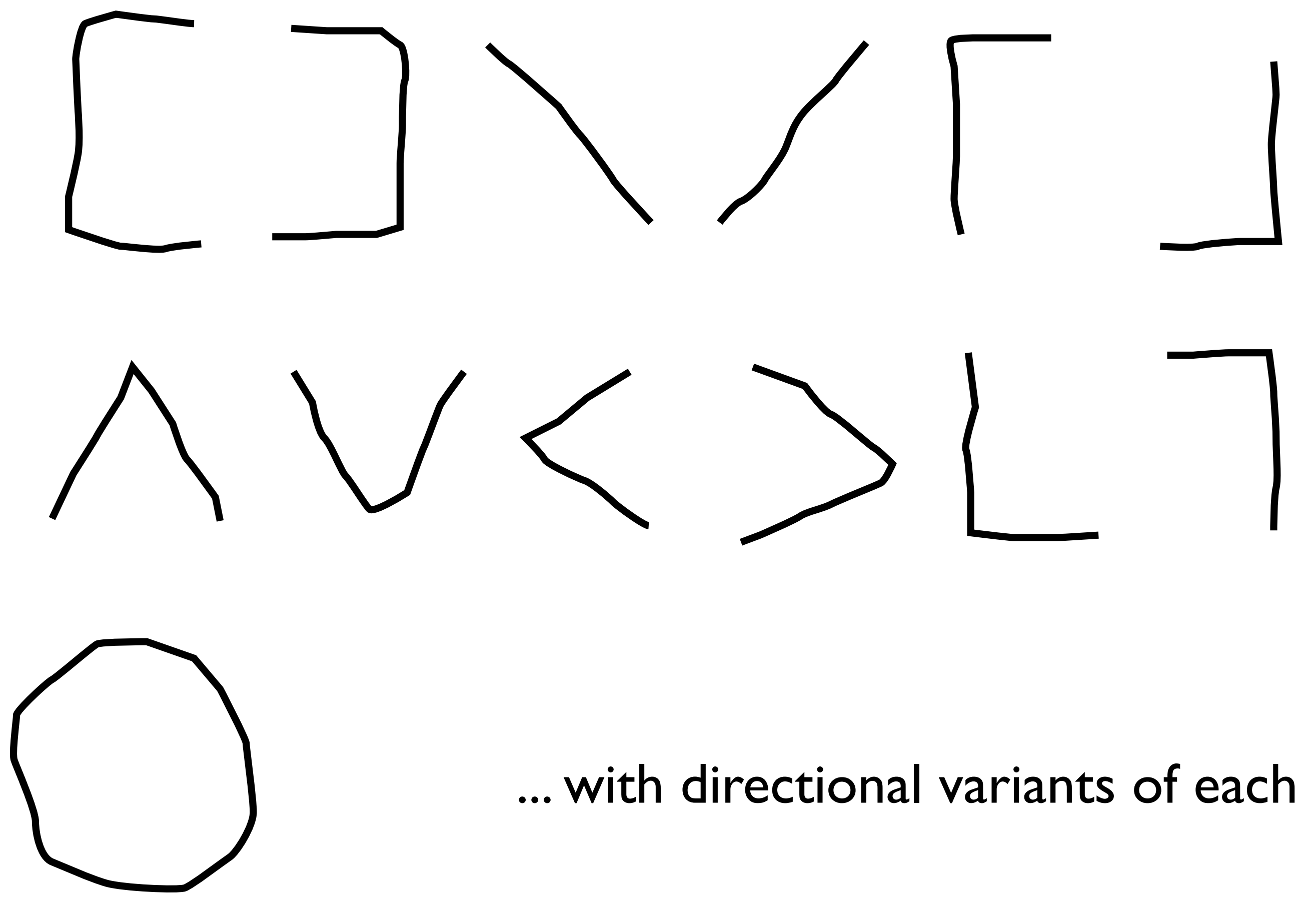


Directional Nine-Squares

- Use an alternative stroke/template representation that preserves ordering across the subsquares
- Example:
  - top-to-bottom: {3, 2, 1, 4, 7, 8, 9}
  - bottom-to-top: {9, 8, 7, 4, 1, 2, 3}
- Can be extended to don't cares also
- (Treat don't cares as wild cards in the matching process)



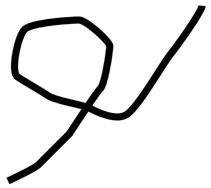
Sample 9-square Gestures



... with directional variants of each

Another Simple Recognizer

- 9-square is great at recognizing a small set of regular gestures
- ... but other potentially useful gestures are more difficult
  - Example: "pigtail" gesture common in proofreaders' marks
- Do we need to go to a more complicated "real" recognizer in order to process these?
- **No!**



The SiGeR Recognizer

- SiGeR: Simple Gesture Recognizer
- Developed by Microsoft Research as a way for users to create custom gestures for Tablet PCs
- Resources:
  - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dntablet/html/tbconCuGesRec.asp
  - http://sourceforge.net/projects/siger/ (C# implementation)

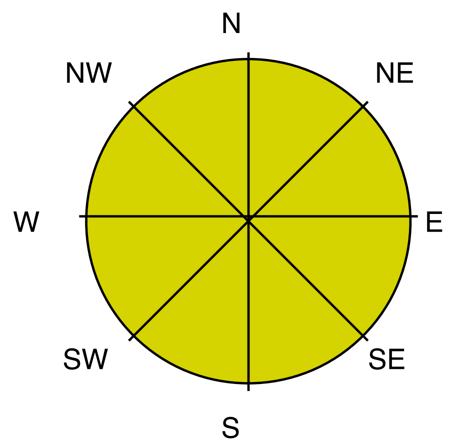
- Big idea:
  - What if you could turn gesture recognition problem into a regular expression pattern matching problem?
  - Reuse existing regexp machinery and turn it into a gesture recognizer!

Basic Algorithm

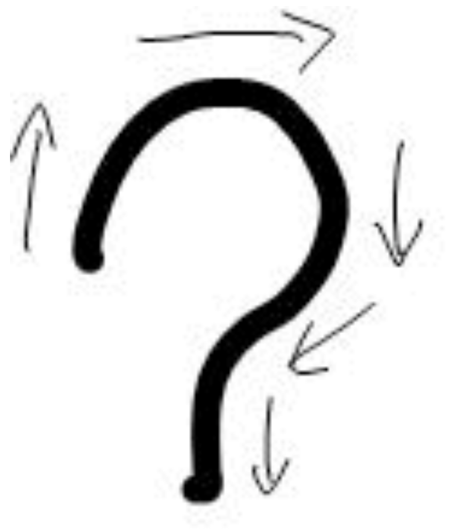
1. Processes successive points in the stroke
2. Compute a direction for each stroke relative to the previous one, and output a vector of symbols representing the directions
3. Define a pattern string that represents the basic shape of the gesture you want to match against
4. Compare the direction vector to the pattern expression; can even use standard regular expression matching

Only One Tricky Part...

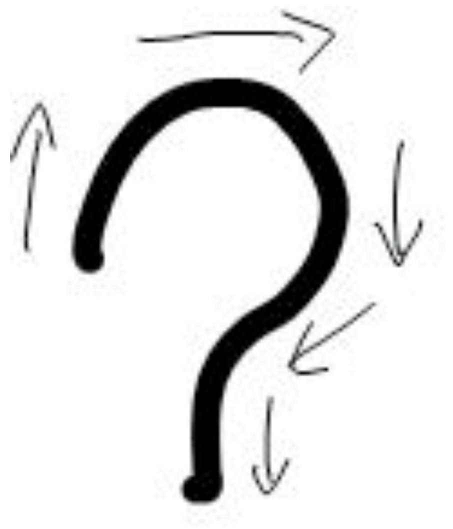
- Getting the representations right to make our job easier when it comes time to match.
- We'll use 8 ordinal directions representing compass points



1. Process Successive Points in the Stroke

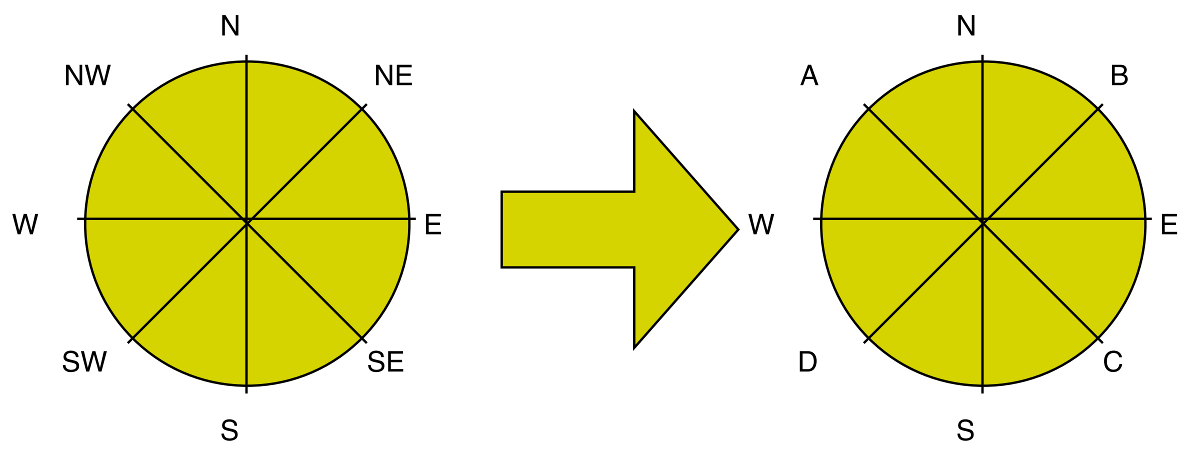


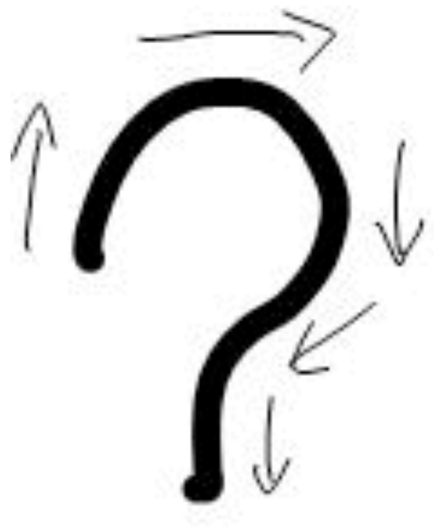
2. Compute a direction vector based on each point



N, N, N, NE, NE, E, E, E, SE, SE, S, S, S, SW, SW, SW, SW, W, S, S, S, S, S

2.a. To make our job easier, rename the directions so we can put them in one big string

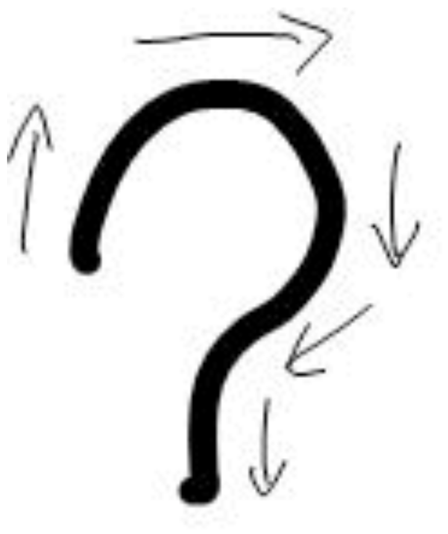




N, N, N, NE, NE, E, E, E, SE, SE, S, S, S, SW, SW, SW, SW, W, S, S, S, S, S

$\Rightarrow$ NNNBBEEECCSSSDDDDDSSSSS

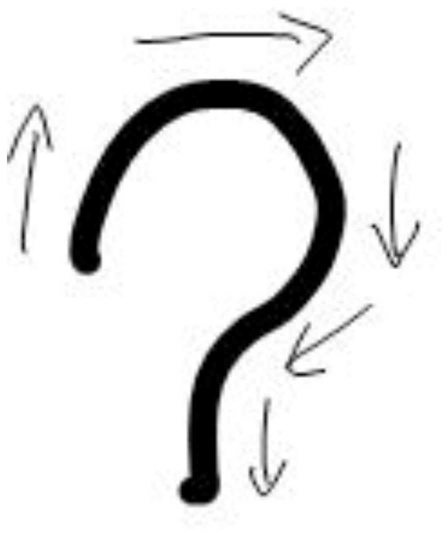
3. Define a pattern string that represents the overall shape of the gesture



NNNBBEEECCSSSDDDDDSSSSS

Question mark is:
•generally up
•then generally right
•then generally down
•then generally left
•then generally down
(defines basic shape of the stroke)

3.a. How to define the template?



Reuse the ordinal direction symbols
N, S, E, W, A, B, C, D

Plus symbols representing more *general* directions
NORTHS = N, NE, NW (N, A, B)
EASTS = E, NE, SE (E, B, C)

Template = [NORTHS, EASTS, SOUTHS, WESTS, SOUTHS]
(defines basic shape of the stroke)

Defining the Template

- Allows you to specify template at greater or lesser specificity
  - Use ordinal symbols when you want a precise match
  - General symbols when you want more "slack"

- The template is then matched against the direction vector by seeing if the template patterns occur

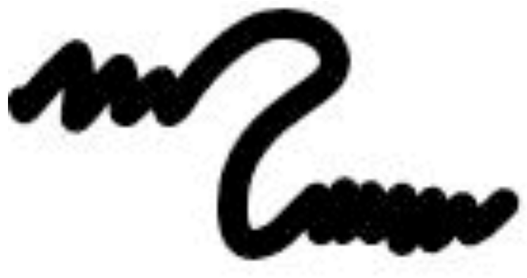
4. How to Match?

- Turn the template vector into a regexp
- See if the pattern is matched in the direction string
- Example:
  - template = [NORTHS, EASTS, SOUTHS, WESTS, SOUTHS]
  - regexp = “[NAB]+[BEC]+[DSC]+[AWD]+[DSC]+”

  - Pattern qm = Pattern.compile(regexp)
  - if (qm.matcher(directionVector).find()) {
    - // it matches!
  - }

How Robust is This?

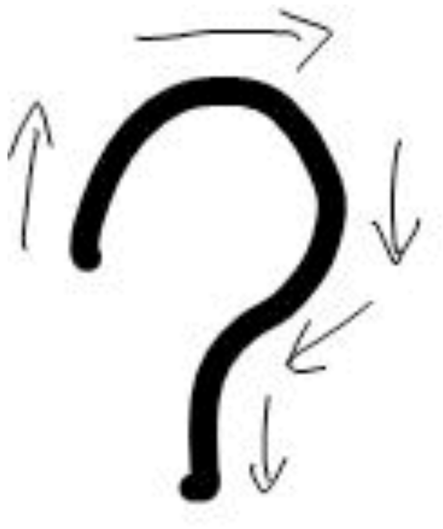
- Here's a gesture that shouldn't match but may, depending on implementation
- Why?
  - A question mark appears in the middle of the stroke



- Therefore:
  - Important to match the **whole** stroke, not just **part** of it!
  - Think of the pattern as including ^ and $ (regular expression markers for beginning of line and end of line) at the first and end

How Robust is This?

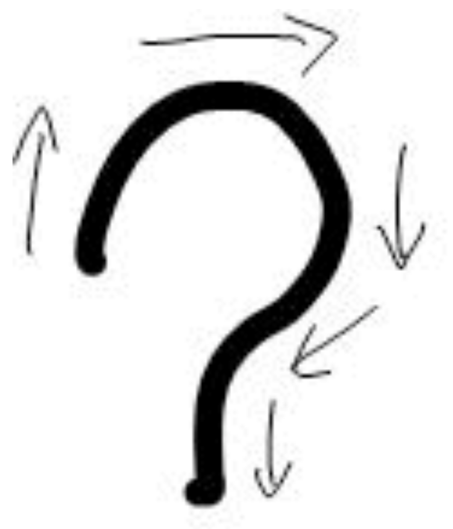
- But requiring the **entire** stroke to match the pattern introduces a new problem
- Can you tell what it is?



How Robust is This?

- But requiring the **entire** stroke to match the pattern introduces a new problem
- Can you tell what it is?

- Look closely at the question mark
  - At the bottom, the stroke jags off to the left
  - Common for the pen to make little tick marks like this when it comes into contact with the tablet, or leaves it



Solution

- Simply trim the beginning and end points of the vector!
- More generally:
    - Ignore small outlier points if the overall shape otherwise conforms to the shape pattern specified in the template.

Implementing SiGeR (one method)

- Specify some helper constants:

```
int UP = (1<<0);
int DOWN = (1<<1);
// ... define other constants, as well as unique tokens that represent
// direction classes
int RIGHT_UP = (RIGHT | UP);
int UPS = (UP | RIGHT_UP | LEFT_UP);
```

- Specify templates in code, using human-readable constants:

```
int QUESTION_MARK = { UPS, RIGHTS, DOWNS, LEFTS, DOWNS };
```

Implementing SiGeR (continued)

- Create a function buildPatternString() that takes the template and emits a regexp pattern that will be used to match it

```
buf.append(“^”);              // match the start of input
buf.append(“.{0,2}+”);        // consume any character 0-2 times (this gets rid of the noise at the beginning)
for (int i=0 ; i<pattern.length ; i++) {
    switch (pattern[i]) {     // emit a unique letter code for each of the 8 directions
        case RIGHT: buf.append(“R+”); break;
        case UP: buf.append(“U+”); break;
        case RIGHT_UP: buf.append(“W+”); break;
        case LEFT_UP: buf.append(“X+”); break;
        // ...
        case UPS: buf.append(“[UWX]+”); break;    // combination directions combine letters
    }
}
buf.append(“.{0,2}+);
buf.append(“$”);
```

Implementing SiGeR (Cont'd)

- Write a function buildDirectionVector() that takes an input stroke and returns a direction vector
  - Compare each point to the point previous to it
  - Emit a symbol to represent whether the movement is UP, RIGHT, etc.
  - (using all of the 8 ordinal directions)
- Use the Java regular expression library to match strokes to patterns!

```
import java.util.regex.*;
if (questionMarkPattern.matcher(strokeString).find()) {
    // it's a question mark!
}
```

More on SiGeR

- SiGeR actually does much more than this; we're just implementing the most basic parts of it here.
- Example: collects statistical information about strokes that can be used to disambiguate them
  - Percentage of the stroke moving right, distance between the start and end points, etc.
  - Can help disambiguate a ring from a square
- Also computes various other features
  - Are shapes open or shut, pen velocity, etc.
  - Can tweak patterns by requiring certain features