PSEUDOCODE FOR THE $Q RECOGNIZER

We provide complete pseudocode for $Q. In the following, POINT is a structure that exposes *x*, *y*, and *strokeId* properties; *strokeId* is the stroke index a point belongs to and is filled by counting touch down/touch up events; *x* and *y* are integers in the set $\{0, 1, ..., m-1\}$, where *m* is the size of the look-up table. POINTS is a list of points and TEMPLATES a list of POINTS with associated gesture class data. The pseudocode assumes that templates have already been preprocessed (when loading the template set, for instance).

$Q-RECOGNIZER (POINTS *points*, TEMPLATES *templates*)

```
n ← 32, m ← 64   // defaults for cloud size (n) and size of the look-up table (m)
NORMALIZE(points, n, m)   // templates have already been normalized
score ← ∞
for each template in templates do
    d ← CLOUD-MATCH(points, template, n, score)
    if d < score then
        score ← d
        result ← template
return ⟨result, score⟩   // the closest template from the set and the smallest score
```

CLOUD-MATCH (POINTS *points*, POINTS *template*, int *n*, int *min*)

```
step ← ⌊n^0.5⌋
// compute lower bounds for both matching directions between points and template
LB_1 ← COMPUTE-LOWER-BOUND(points, template, step, template.LUT)
LB_2 ← COMPUTE-LOWER-BOUND(template, points, step, points.LUT)
for i ← 0 to n − 1 step step do
    if LB_1[i/step] < min then
        min ← MIN(min, CLOUD-DISTANCE(points, template, n, i, min))
    if LB_2[i/step] < min then
        min ← MIN(min, CLOUD-DISTANCE(template, points, n, i, min))
return min
```

CLOUD-DISTANCE (POINTS *points*, POINTS *template*, int *n*, int *start*, float *minSoFar*)

```
unmatched ← {0, 1, 2, ..., n − 1}   // indices of unmatched points from template
i ← start      // start the matching from this index in the points cloud
weight ← n   // weights decrease from n to 1
sum ← 0      // computes the cloud distance between points and template
do
    min ← ∞
    for each j in unmatched do
        d ← SQR-EUCLIDEAN-DISTANCE(points[i], template[j])
        if d < min then
            min ← d
            index ← j
    REMOVE(unmatched, index)   // implementable in O(1)
    sum ← sum + weight · min
    if sum ≥ minSoFar then
        return sum          // early abandoning of computations
    weight ← weight − 1   // weights decrease from n to 1
    i ← (i + 1) MOD n     // advance to the next point in points
until i == start
return sum
```

COMPUTE-LOWER-BOUND (POINTS *points*, POINTS *template*, int *step*, int[,] *LUT*)

```
LB ← new float[n/step + 1]   // multiple lower bounds, one for each starting point
SAT ← new float[n]   // summed area table for fast computations (see text)
// first, compute the lower bound for starting point index 0
LB[0] ← 0
for i ← 0 to n − 1 do
    index ← LUT[points[i].x, points[i].y]
    d ← SQR-EUCLIDEAN-DISTANCE(points[i], template[index])
    SAT[i] ← (i == 0) ? d : SAT[i−1] + d
    LB[0] ← LB[0] + (n − i) · d
// compute the lower bound for the other starting points (see formula in the text)
for i ← step to n − 1 step step do
    LB[i/step] ← LB[0] + i · SAT[n−1] − n · SAT[i−1]
return LB
```

The following pseudocode implements gesture preprocessing: resampling, translation to origin, rescaling into the $m \times m$ grid, and computation of the look-up table. Except for the new COMPUTE-LUT function and changes in the SCALE function, this pseudocode is practically the same as for $P [50] (p. 280).

NORMALIZE (POINTS *points*, int *n*, int *m*)

```
points ← RESAMPLE(points, n)
TRANSLATE-TO-ORIGIN(points, n)
SCALE(points, m)
LUT ← COMPUTE-LUT(m, points, n)
```

RESAMPLE (POINTS *points*, int *n*)

```
I ← PATH-LENGTH(points) / (n − 1)
D ← 0
newPoints ← {points[0]}
for i ← 1 to n − 1 do
    if points[i].strokeId == points[i−1].strokeId then
        d ← EUCLIDEAN-DISTANCE(points[i−1], points[i])
        if (D + d) ≥ I then
            q.x ← points[i−1].x + (I − D)/d · (points[i].x - points[i−1].x)
            q.y ← points[i−1].y + (I − D)/d · (points[i].y - points[i−1].y)
            APPEND(newPoints, q)
            INSERT(points, i, q) // q will be the next points[i]
            D ← 0
        else D ← D + d
return newPoints
```

TRANSLATE-TO-ORIGIN (POINTS *points*, int *n*)

```
c ← (0, 0)   // will compute the centroid of the points cloud
for each p in points do
    c ← (c.x + p.x, c.y + p.y)
c ← (c.x/n, c.y/n)
for each p in points do
    p ← (p.x − c.x, p.y − c.y)
```

SCALE (POINTS *points*, int *m*)

```
x_min ← ∞, x_max ← −∞, y_min ← ∞, y_max ← −∞
for each p in points do
    x_min ← MIN(x_min, p.x)
    y_min ← MIN(y_min, p.y)
    x_max ← MAX(x_max, p.x)
    y_max ← MAX(y_max, p.y)
s ← MAX(x_max − x_min, y_max − y_min)/(m − 1)   // scale factor
for each p in points do
    p ← ((p.x − x_min)/s, (p.y − y_min)/s)   // p.x and p.y are now integers in 0...m − 1
```

COMPUTE-LUT (POINTS *points*, int *n*, int *m*)

```
LUT ← new int[m, m]
for x ← 0 to m − 1 do
    for y ← 0 to m − 1 do
        min ← ∞
        for i ← 0 to n − 1 do
            d ← SQR-EUCLIDEAN-DISTANCE(points[i], new POINT(x, y))
            if d < min then
                min ← d
                index ← i
        LUT[x, y] ← index
return LUT
```

PATH-LENGTH (POINTS *points*)

```
d ← 0   // will compute the path length
for i ← 1 to n − 1 do
    if points[i].strokeId == points[i−1].strokeId then
        d ← d + EUCLIDEAN-DISTANCE(points[i−1], points[i])
return d
```

SQR-EUCLIDEAN-DISTANCE (POINT *a*, POINT *b*)

```
return (a.x − b.x)^2 + (a.y − b.y)^2   // much faster to compute without the sqrt()
```

EUCLIDEAN-DISTANCE (POINT *a*, POINT *b*)

```
return sqrt(SQR-EUCLIDEAN-DISTANCE(a, b))
```