Motion Localization

Grayscale Image Localization

Phase

Compare Frames

Compare Frames

Compare image matrices and mark the pixel location that have two significantly different values

Result

Check Diagonals

Check Diagonals

Set coordinates to bound the clusters

Result

Cluster Find

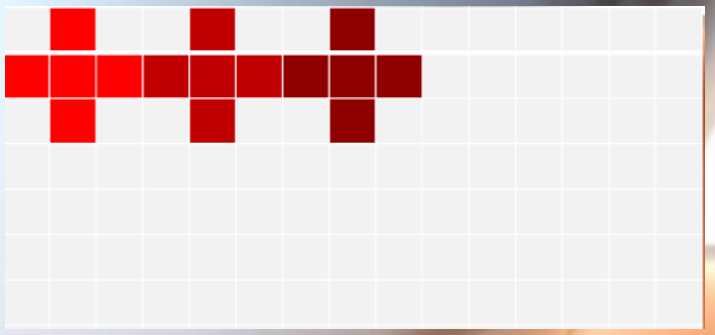
Cluster Find

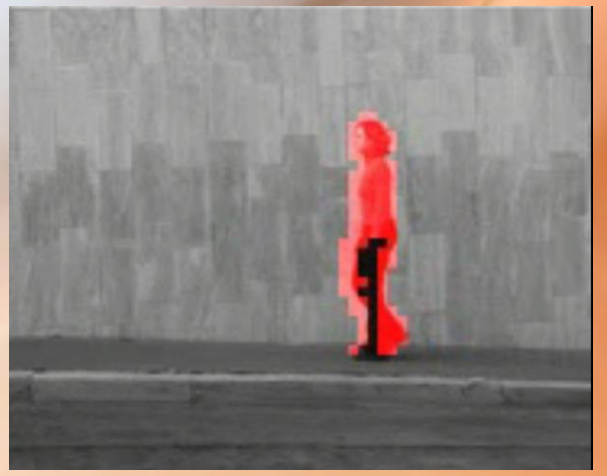
Checks for overlapping rectangles and chooses the largest

Result

Pseudocode - Pixel Compare

```
Input Frame_Library //Images in exercise folder
Input pixel_rows       //Pixels in x-axis in image
Input pixel_columns //Pixels in y-axis
Input threshold        //Amount of pixel difference to determine change
Output Frame2
// width = W | height = H | kernel Diameter = D  | W*H = n
Frame1 = frame_library[f]                                          // 1
Frame2 = frame_library[f+1]                                        // 1
    For i ← 0 in pixel_rows ; i += D                               // W*1/D
        For j ← 0 in pixel_columns; j += D                         // W*H*(1/D)*(1/D)
            mean_condition   ←   (absolute(frame2(i - 1, j) - frame1(i - 1, j) +
            absolute( frame2(i, j + 1) - frame1(i, j + 1)) +
            absolute(frame2 ( i  , j - 1) - frame1(i, j-1)) +
            absolute(frame2( i + 1 , j ) - frame1( i + 1 , j )) +
            absolute(frame2( i , j ) - frame1( i , j)))/5
                                                                   // W*H*(1/D)*(1/D)*10
                                                                     (10 assignments)
            if mean_condition > threshold                          // W*H*(1/D)*(1/D)
                for x ← 0 in kernel_rows                           // W*H*(1/D)*(1/D)*D
                    for y ← 0 in  kernel_columns                   // W*H*(1/D)*(1/D)*D*D
                        frame2( i+x , j+y) = 255
                                                                   // ( L-1)*W*H*(1/D)*(1/D)*D*D
 Return Frame2                                                     // (L–1)
```





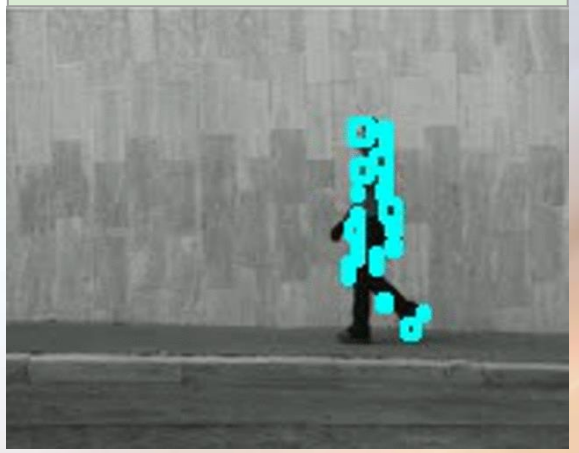
```
// 3 + W*(1/D)  +  12*W*H*(1/D)*(1/D)
//+ W*H*(1/D)*(1/D)*R + 2* W*H*(1/D)*(1/D)*D*D
//Taking the biggest term:
//2* W*H*(1/D)*(1/D)*D*D =  2* W*H -2*W*H
//2* W*H = 2*n  => O(n)
```

Pseudocode - Check Diagonals

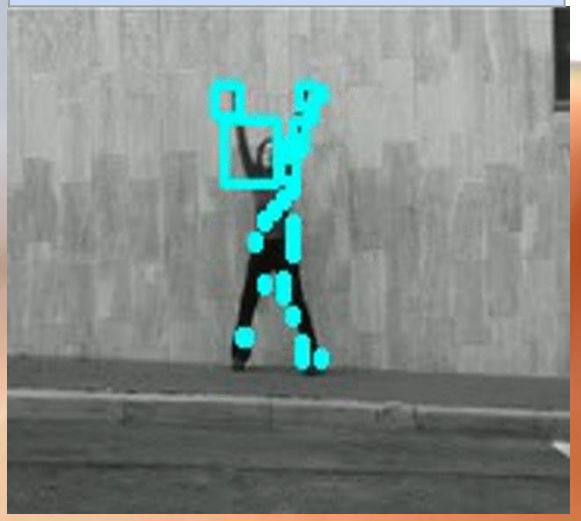
```
Input Frame_Changes              // Image with significant changes highlighted in red
Input threshold                  // Area of pixels to ignore
Input total rows , total columns // width and height of the image
Output rectangle_list            //list of rectangles
//width = W | height = H | W*H = n
CheckDiagonals(Frame_Changes ,total_rows ,total_column, threshold):

rectangle_list = []                                                    //1
For i ← 0 in total_rows                                                // H
    For j ← 0 in total_columns                                         //H*W
        width = 0                                                      // H*W
        height = 0                                                     // H*W
        no_repeat = True                                               // H*W
        coord_stack = ([ j , I ])                                      // H*W
        while (len(coord_stack) > 0 and x < total_columns and y < total_rows):  // H*W (W+H)
            x,y = coord_stack.pop()                                    // H*W (W+H)
            if frame[x+1,y,2] != 255 and frame[x,y+1,2] !=255:         // H*W*(2*(W+H))
                continue                                               //H*W
            if frame[x+1,y,2] ==255 and no_repeat:                     //2*(H*W)*(W+H)
                no_repeat = False                                      //W*(H*W)
                height+=1                                              //W*(H*W)
                coord_stack.push(x+1,y)                                //W*(H*W)
            else:                                                      //W*(H*W)
                no_repeat = True                                       //H*(H*W)
                width +=1                                              //H*(H*W)
                coord_stack.push(x,y+1)                                //H*(H*W)
        if (width + height) > threshold                                // H*W
            rectangle = j, I , j+ width, i + height                    // H*W/threshold
            rectangle_list.append(rectangle)                           // H*W/threshold
return rectangle_list                                                  //1
```

Exercise 2



Exercise 1



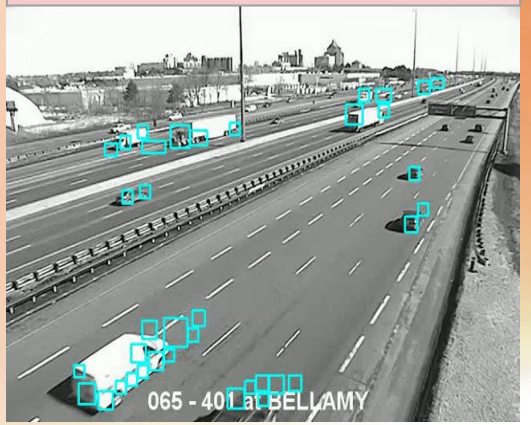
```
//2 + H + 7*H*W + 2*H*W/threshold + 9*(H*W)*(H+W)
//Taking the biggest term
//9*n*(H+W) → 9*n*log(n) → O(n*log(n))
```

Pseudocode - Cluster Bounding

```
Input Rectangle_List          //list of rectangle points
Input Overlap_Threshold     // Overlap allowance (area allowed)
Output Boxes                  // Remaining list of rectangle points
Cluster_find(Rectangle_list , Overlap_threshold):
    if len(Rectangle_list) = 0                                                    //1
        return Rectangle_list                                                     // 0
    x1 = rectangle_List[for each x,0], x2 = rectangle_List[for each x,1] , y1 = rectangle_List[ for each y, 2] ,
            y2 = rectangle_List[ foreach,3]                                       //4n
    Index = QuickSortByIndex(y2)                                                  //n^2 (quick sort)
    area = (x2-x1) * (y2-y1)                                                      //1
    while len(index) > 0:                                                         //n+1
        last = len(index) -1                                                      //n
        i=index[last]                                                             //n
        pick.append(i)                                                            //n
        big_x1 = Min(x1[i], x1[index[for each last]])                             //n+(n-1)+ ...+1
        big_x2 = Max(x2[i], x2[index[for each last]])                             // n+(n-1)+ ...+1
        big_y1 = Min(y1[i], y1[index[for each last]])                             // n+(n-1)+ ...+1
        big_y2 = Max(y2[i], y2[index[for each last]])                             // n+(n-1)+ ...+1
        width = Max(0,big_x2 -  big_x1)                                           //n
        height = Max(0,big_y2 - big_y1)                                           // n
        overlap = (w*h)/area[index[for each last]]                                //n + (n-1)+ ...+1
        index = delete(index, concatenate([last], where ( overlap > Overlap_Threshold ))   //n +(n-1) + ... +1
    return    rectangle_list[pick]                                                //1
//7 +  n^2 +   7*n + 5*n(n-1)/2 →    5*n(n-1)/2 → O(n²)
```

Exercise 3

Exercise 4

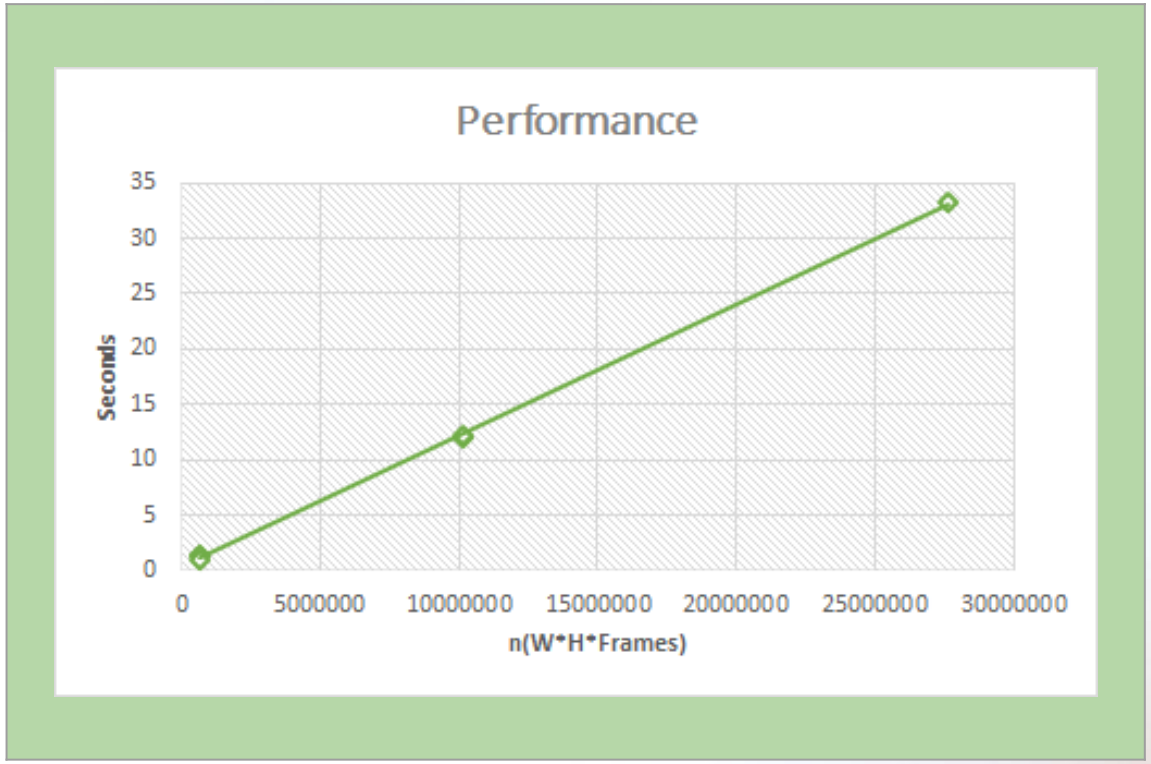


Timing Breakdown - After Grouping

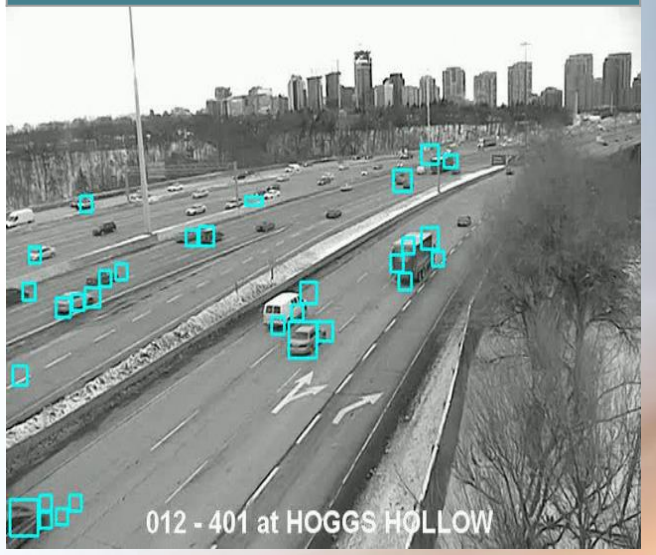
Exercise	Resolution	Frames	Time	FPS
Exercise 1	180 x 144	25	1.42s	17.6
Exercise 2	180 x 144	25	1.35s	23.7
Exercise 3	180 x 144	25	0.938s	26.7
Exercise 4	704 x 480	30	11.99s	2.5
Exercise 5	704 x 480	30	12.16s	2.5
Exercise 6	640 x 480	90	33.23s	2.7

- Time illustrated is time it takes to process all frames in the exercise
- FPS = Frames per Second

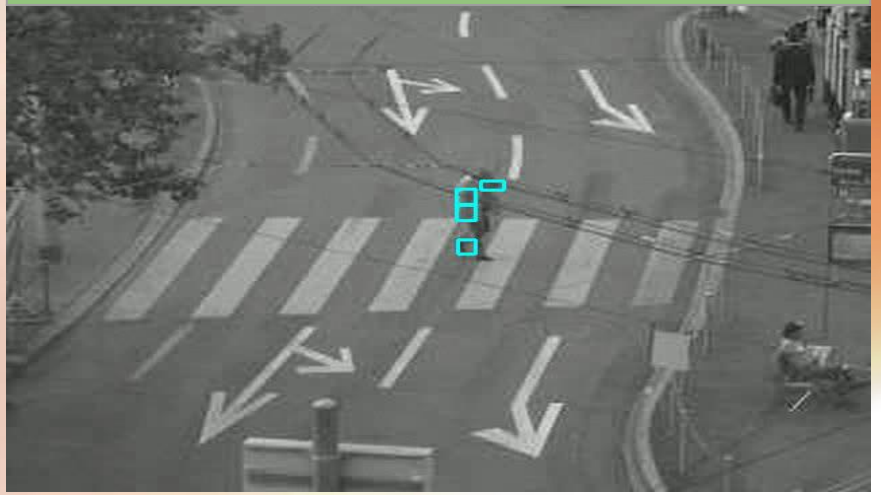
Performance Plot



Exercise 5



Exercise 6

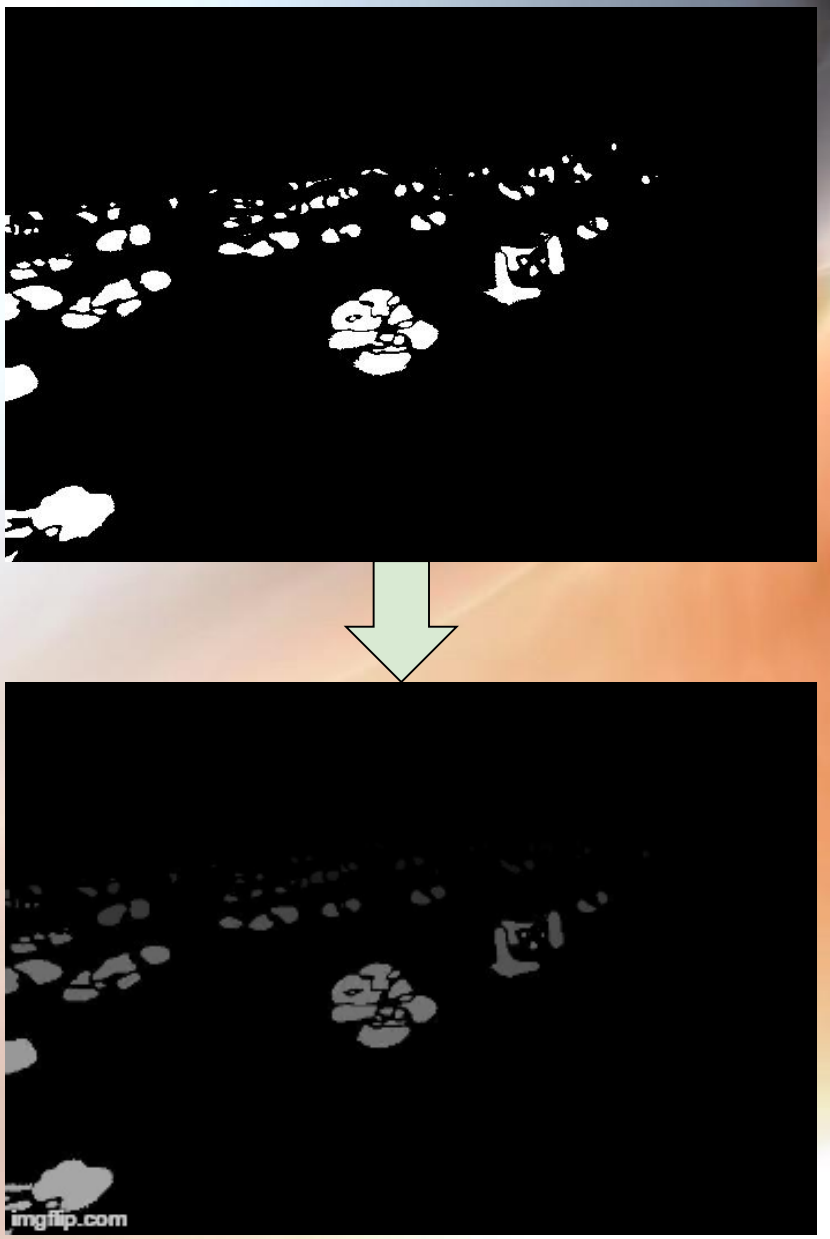


- While the Big O of our program is n^2 , it seems that in practice it appears linear

Pseudocode - Connected Components

Connected components with Union-Find
This function utilizes previously created functions to create an image of clusters with each cluster having its own unique label
INPUTS
B is the original binary image
LB will be the labeled connected component image, initially matrix same size as **B** but filled with zeros
OUTPUT
LB labeled connected component image

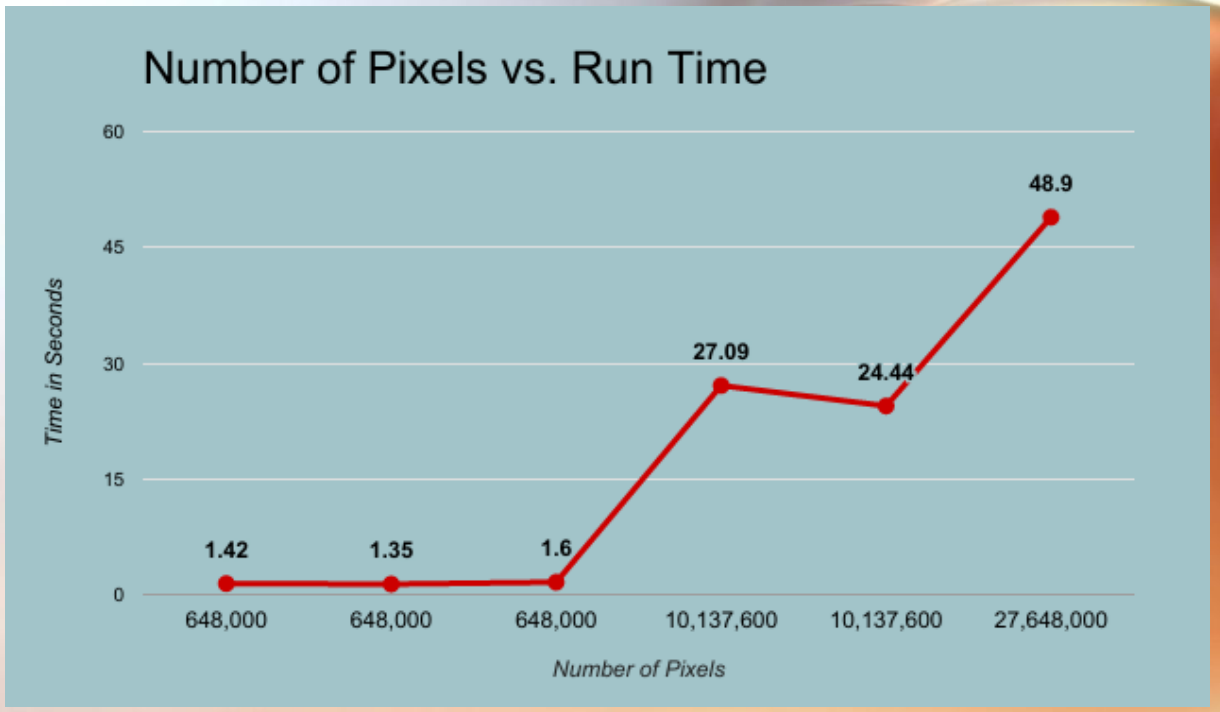
```
Procedure classical_with_union-find(B, LB)
Initialize()
for L <- 0 to MaxRow
                                        //x
                    for P <- 0 to MaxCol
                                        //y
                                        if B[L,P] == 1 then
                                        //(x*y)=n
                                        A <- prior_neighbors(L,P)
                    //n
                                        if isempty(A) then
                                        //n
                                                            M <- label
                                        //n
                                                            label <- label +1
                                        //n
                                        else
                                                            //n
                                                            M <- min(labels(A))
                                        // 4*n
                                        LB[L,P]  <- M
                                        //n
                                        for X in labels(A)
                                        //4*n
                    if X != M
```



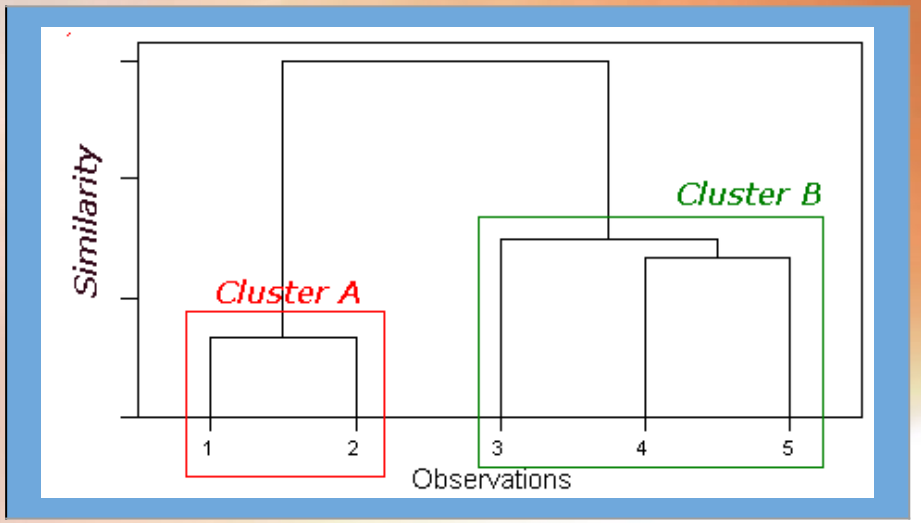
Timing Breakdown - Connected Components

Exercise	Resolution WxHxF	Input Pixels	Time	Cars
Exercise 1	180 x 144 x 25	648,000	1.97s	N/A
Exercise 2	180 x 144 x 25	648,000	1.986s	N/A
Exercise 3	180 x 144 x 30	648,000	1.604s	N/A
Exercise 4	704 x 480 x 30	10,137,600	27.1s	36
Exercise 5	704 x 480 x 30	10,137,160	24.44s	40
Exercise 6	640 x 480 x 90	27,648,000	48.9s	N/A

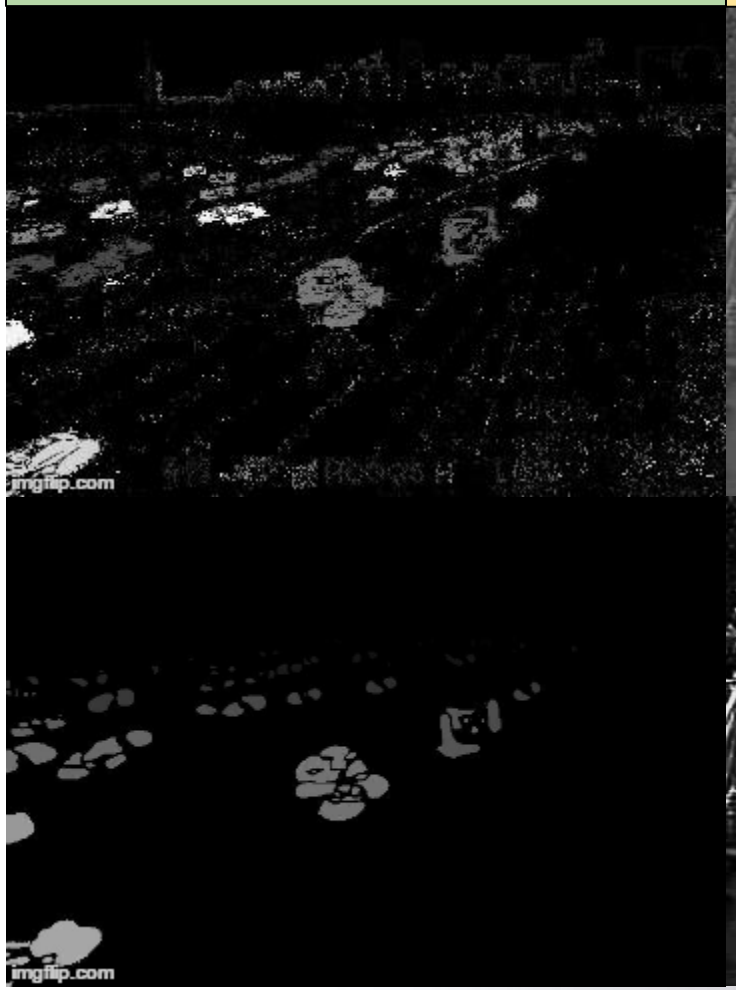
W = Width
H = Height
F = Frames



Component Grouping Algorithm



More on Image Processing

Gaussian Filtering	Histogram Equalization
	

Gaussian Filter
- Removes noise
- Leaves useful data
- Emphasizes large objects

Histogram Equalization
- Increases the contrast between pixels
- Easier to detect changes