# Group: Die Ouens

# Members:


## Yaseen Haffejee 1827555

## Ziyaad Ballim 1828251

## Jeremy Crouch 1598024

## Fatima Daya 1620146


# Importing Libraries

In [1]:

```
import numpy as np
import math
import cv2
import json
import imageio
from skimage import img_as_float32 as as_float
from natsort import natsorted
from glob import glob
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
import networkx as nx
import random
```

In [2]:

```
## This code was given
path_pairs = list(zip(
natsorted(glob('./puzzle_corners_1024x768/images-1024x768/*.png')),
natsorted(glob('./puzzle_corners_1024x768/masks-1024x768/*.png')),
))

imgs = np.array([as_float(imageio.imread(ipath)) for ipath, _ in path_pairs])
msks = np.array([as_float(imageio.imread(mpath)) for _, mpath in path_pairs])
```

# 1) Find Contours

## 1.1)

1 Write a function called get_puzzle_contour(mask) that takes in a binary mask of a puzzle piece and returns a single contour array of that puzzle piece. A list of contours in a mask can be found using cv2.findContours . Since multiple contours are returned, assume that the correct contour corresponding to the boundary of the puzzle piece is closed and the one with the largest area

In [3]:

```
def get_puzzle_contour(mask):

    contours, hierarchy  = cv2.findContours(mask.astype(np.uint8), cv2.RETR_LIST, cv2.CHAIN

    closed_contours = []

    for contour in contours:
        if cv2.contourArea(contour) > cv2.arcLength(contour, True):
            closed_contours.append(contour)

    sorted_contours = sorted(closed_contours, key=cv2.contourArea, reverse=True)

    final_contour = sorted_contours[0]
    final_contour = final_contour.reshape((1,final_contour.shape[0], final_contour.shape[2]
    return final_contour
```

## 1.2)

To ensure we can tell if the side of a puzzle piece is sunken or protruding, later on, we need the points in the contour to be ordered in a clockwise direction. Code a new function called get_clockwise_contour(contour) that returns a copy of the contour, but reversed if points in the contour are ordered anti-clockwise

In [4]:

```
def get_clockwise_contour(contour):
    area = cv2.contourArea(contour,oriented=True)

    if area < 0:
        contour = contour[::-1]
    return contour
```

## 1.3)

Tino doesn't like the idea of using libraries if we don't know what they are actually doing under the hood. Do some research on how cv2.findContours works as well as how the oriented=True version of cv2.contourArea detects orientation. Give a short (maximum one paragraph) explanation for each.

cv2.findContours

- This function works by finding the boundaries of the objects in a binary image where the background is black and foreground is white. It find the contours by finding the points where there is a change from intensity 1 to 0 (ie foreground to background).

cv2.contourArea(oriented=True)

- This function returns the area of the region contained within the contour. When given the flag oriented= True, it will return a negative value if the points in the contour are oriented counterclockwise, and a positive value if they're clowckwise.

## 1.4)

Finally plot the result of any three reoriented boundaries (contours) overlaid on top of their corresponding original RGB puzzle piece images. Do not use the same puzzle pieces as the ones used in figure 1 and 2

In [5]:

```
images_to_plot = [14,15,16]
# images_to_plot = [8,22,31]
figure_contours = plt.figure()
# figure_contours = plt.figure(figsize = (8,30))
plt.axis("off")
plt.title("Contours")
figure_contours.set_figheight(8)
figure_contours.set_figwidth(30)
c = 1
for i in images_to_plot:
    contours = get_puzzle_contour(msks[i])
    clockwise_contours = get_clockwise_contour(contours)
    rgb_image = imgs[i].copy()
    cv2.drawContours(rgb_image, clockwise_contours, -1, (0,1,0), 3)
    figure_contours.add_subplot(1, 3, c)
    plt.title(f"Image Number {i}")
    plt.axis("off")
    plt.imshow(rgb_image)
    c+=1

plt.show()
```


# 2) Shape Models

## 2.1) Extract Sides

### 2.1.1)

Load the JSON corner data for each puzzle piece using the python json package. The format of the data is two lists with corresponding elements, the first containing names of the original RGB images, the second containing float arrays of shape (4, 2) that correspond to the 4 corner points. The points in the corner arrays indicate coordinates (x, y) like the contours from cv2, not (y, x) as usual in python. Each x and y coordinate is actually the ratio along that axis of an image, thus, to obtain the original pixels of the corners, multiply each ratio respectively by the width and the height of the target image.

In [6]:

```
### THis code was given partyly
def get_corners(path,height,width):

    with open(path, mode="r") as f:
        names, corner_ratios = json.load(f)

    original_pixels = np.empty_like(corner_ratios)

    corner_width = len(corner_ratios)
    corner_height = len(corner_ratios[0])

    for idx in range(corner_width):
        for y in range(corner_height):
            original_pixels[idx][y][0] = corner_ratios[idx][y][0] * width
            original_pixels[idx][y][1] = corner_ratios[idx][y][1] * height

    return original_pixels
```

In [7]:

```
height, width = msks[0].shape
original_pixels = get_corners("puzzle_corners_1024x768/corners.json",height,width)
```

## 2.1.2)

Next we need to extract the four side contours for each puzzle piece using its contour and four ground truth corners. Code a function called extract_sides(contour, corners) that returns a list of side contours after the following steps:

- Find the index of the nearest point in the contour to each corner point using euclidean distance. Call this list of four indices corner_indices. Ask yourself why using the approximation mechanism cv2.CHAIN_APPROX_SIMPLE in question 1.1 might help here?
- Sort corner_indices in case the 4 indices are not consecutive due to matching, as we need them to correspond to a valid closed contour.
- Construct the 4 non-closed contours for each side of the puzzle piece. The first and last points in these side contours can be obtained using paired indices from corner_indices, with all the points along the contour between these corner points included. Make sure to preserve the clockwise nature of the points along the contour from question 1.2. The list of sides should have the property that indexing for side n should mean that side n+2 opposite it, and n + 1 is to the right or clockwise.

In [8]:

```
def extract_sides(contour,corners):

    corner_indices = []

    for corner in corners:
        corner_dists = np.linalg.norm(contour - corner, axis=-1)
        corner_indices.append(np.argmin(corner_dists))

    corner_indices = np.array(corner_indices[::-1])

    all_sides = []

    for side in range(4):
        i, j = corner_indices[[side, (side+1)%4]]
        pts = np.roll(contour.reshape(-1,2), shift=-i, axis=0)
        sides = pts[0:j-i+1]
        all_sides.append(np.array([sides]))

    return all_sides[::-1]
```

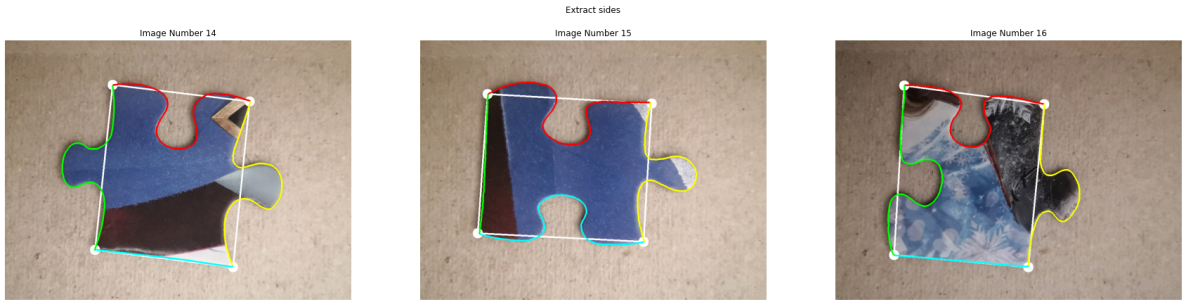
### 2.1.3)

As you did for question 1.4 with the same puzzle pieces, plot the quad formed by the corners with the colour (255,255,255), overlaid by the points of the 4 sides extracted, colouring the different sides with red (255,0,0), green (0,255,0), blue (0,64,255), yellow (255,192,0), do not use the same puzzle pieces as in figure 1 and 2.

In [9]:

```
colours = [(255,0,0),(255,192,0),(0,64,255),(0,255,0)]
white = (255,255,255)
extract_sides_fig = plt.figure()
extract_sides_fig.set_figheight(8)
extract_sides_fig.set_figwidth(30)
plt.axis("off")
plt.title("Extract sides")
c = 1
for i in images_to_plot:
    # Get the contours
    contours = get_puzzle_contour(msks[i])
    # Get the clockwise oriented contours
    clockwise_contours = get_clockwise_contour(contours)
    # Get the contours for the 4 sides
    side_contours = extract_sides(clockwise_contours, original_pixels[i])
    # Get the corners
    corners = np.int32(original_pixels[i]).reshape((-1,1,2))
    # Create a copy of the image
    rgb_img = imgs[i].copy()
    # Plot the corners in white
    cv2.polylines(rgb_img, [corners], True, white, 3)
    cv2.circle(rgb_img, corners[0][0], 15, white, -1)
    cv2.circle(rgb_img, corners[1][0], 15, white, -1)
    cv2.circle(rgb_img, corners[2][0], 15, white, -1)
    cv2.circle(rgb_img, corners[3][0], 15, white, -1)
    # Plot the side contours in different colours
    cv2.polylines(rgb_img, side_contours[0], False, colours[0] , 3)
    cv2.polylines(rgb_img, side_contours[1], False, colours[1], 3)
    cv2.polylines(rgb_img, side_contours[2], False, colours[2], 3)
    cv2.polylines(rgb_img, side_contours[3], False, colours[3], 3)
    extract_sides_fig.add_subplot(1, 3, c)
    plt.title(f"Image Number {i}")
    plt.axis("off")
    plt.imshow(rgb_img)
    c+=1
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




## 2.2) Normalising sides

## 2.2.1)

Code a function called transform_puzzle_side(contour) that takes in a single side contour and returns the translated, scaled and rotated version to be used for matching. Imagine there is an arbitrary line running from the first point 0 to the last point −1 in the contour, we want to find the transformations to translate this line so that it is zero centred, then scale it so that it is of length 2, then rotate it so the line is on the x-axis with the first point pointing along the negative x-axis or towards the left. After applying these transformations6 to the entire side contour, the first point at index 0 should have the coordinate (x, y) == (-1, 0), and the last point at index −1 should have the coordinate (x, y) == (1, 0). Sunken sides should point up while protruding sides should point down. Note that we don't try rotate sunken sides by 180◦ so that they correspond to protruding sides, as this separation between sunken and protruding sides is very useful for matching with knn! If you look at matching sides of puzzle pieces in figure 2, you can verify this final transform. Note too that we would also need to reverse the order of points in the sunken array if we wanted them to correspond. Ask yourself why this is.

In [10]:

```
def transform_puzzle_side(contour):
    reshaped =  np.float64(contour.copy().reshape((-1,2)))

    #Translate/shift
    contour_reshaped = np.float64(contour.reshape(-1,2))
    translated = contour_reshaped.copy()
    translated[:,0] = np.round(translated[:,0] - np.mean(translated[:,0]))
    translated[:,1] = np.round(translated[:,1] - np.mean(translated[:,1]))

    first_pt_x = contour_reshaped[0,0]
    first_pt_y = contour_reshaped[0,1]
    last_pt_x = contour_reshaped[-1,0]
    last_pt_y = contour_reshaped[-1,1]

    translation_x = (last_pt_x-first_pt_x)/2.0
    translation_y = (last_pt_y-first_pt_y)/2.0

    shifted = np.float64(contour_reshaped.copy())
    shifted[:,0] -= (translation_x + first_pt_x)
    shifted[:,1] -= (translation_y + first_pt_y)

    #Scale
    max_value = max(max(abs(shifted[:,0])), max(abs(shifted[:,1])))
    scaled = np.float64(shifted.copy())
    scaled /= max_value

    #Rotated
    point1_x, point1_y = scaled[-1]
    theta = np.arctan2(point1_y, point1_x)

    rotated = np.float64(scaled.copy())
    rotated[:,0] = np.cos(theta)*scaled[:,0] + np.sin(theta)*scaled[:,1]
    rotated[:,1] = -np.sin(theta)*scaled[:,0] + np.cos(theta)*scaled[:,1]

    rotated *= -1

    return rotated
```

## 2.2.2)

Finally when normalising the sides for matching, we need an equal number of evenly spaced points along the side to be flattened into a feature vector. More advanced interpolation might work better, but for now, we just use linear interpolation over the entire length of a non-closed contour. Code a new function called even_spaced_contour(contour, num_points=64) that takes in a contour and the number of points that should be contained in the returned contour, then performing the following steps:

- To perform the linear interpolation we can use Bnew = np.interp(Anew, A, B), for clarity our argument names differ from the docs. np.interp works by looking at corresponding points (ai , bi) where ai ∈ A, bi ∈ B and finding the new points (a 0 j , b0 j ) where a 0 j ∈ Anew and b 0 j is the linearly interpolated value.
- To find A we need to calculate the cumulative sum of the length of all the line segments in the contour, divided by the total length of the contour. Note that the length of this array should be equal to the number of points, insert 0 at the front of the array returned by np.cumsum, since the number of segments is 1 less than the number of points.
- Once we have these cumulative ratios A for each point along the length of the line, we generate Anew = np.linspace(0, 1, num_points) as the evenly spaced values between 0 and 1.
- Apply np.interp to all x values then all y values, with B = (x0, ..., xn) and then B = (y0, ..., yn) obtained from our contour points. recombine the new x and y values into your new evenly spaced contour points.

In [11]:

```
def even_spaced_contour(contour, num_points=64):
    reshaped = np.float64(contour.copy().reshape((-1,2)))

    segment_lengths = []

    for i in range(reshaped.shape[0]-1):
        length = np.linalg.norm(reshaped[i] - reshaped[i+1], axis=-1)
        segment_lengths.append(length)

    cumulative_sum = np.cumsum(segment_lengths)

    A= np.zeros((cumulative_sum.shape[0]+1))
    A[1:] = cumulative_sum
    A /= A[-1]

    A_new = np.linspace(0,1,num_points)

    B_new = np.zeros((num_points, 2))
    B_new[:,0] = np.interp(A_new, A, reshaped[:,0])
    B_new[:,1] = np.interp(A_new, A, reshaped[:,1])


    B_new[0] = reshaped[0]
    B_new[-1] = reshaped[-1]

    return np.int64(B_new)
```

## 2.2.3)

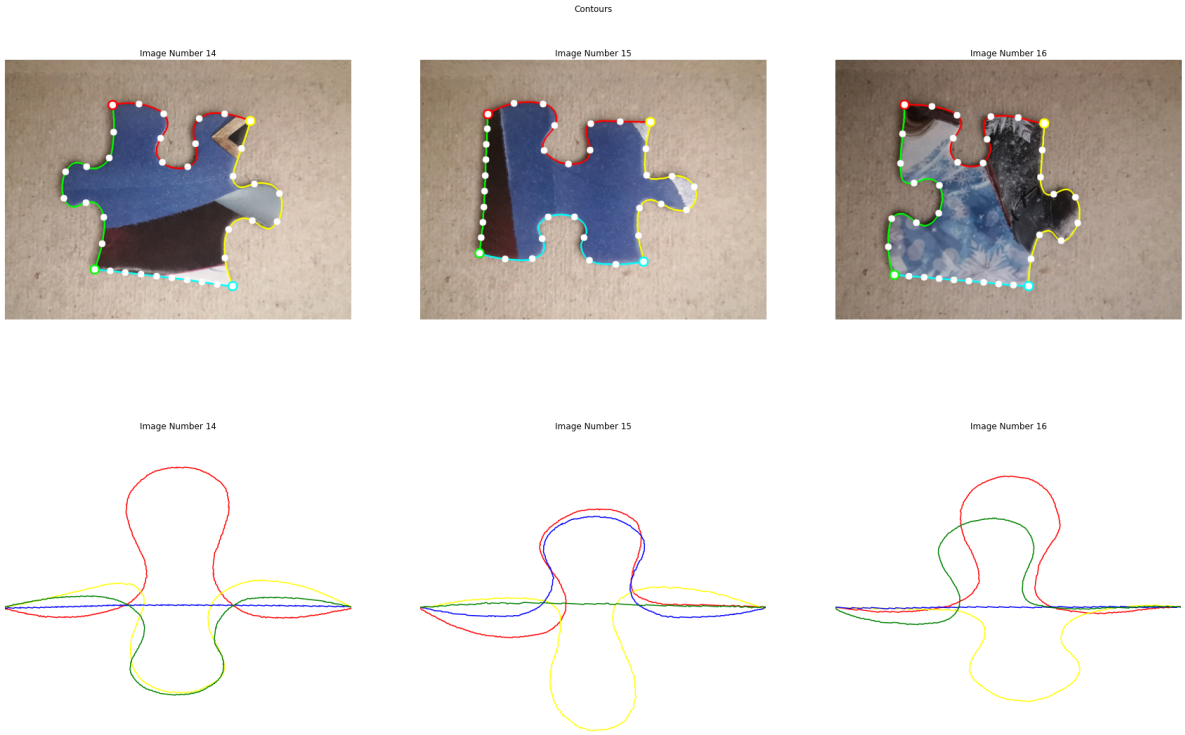
As you did for question 2.1.3 with the same puzzle pieces, plot the new normalised sides with the same colours centred on a blank image as in figure 2. Do not use the same puzzle pieces as in figure 1 and 2. Overlay this plot with points obtained from the simplified versions of those same normalised side contours after applying even_spaced_contour with num_points=10 (remember to use 64 or more for the rest of the lab). Make sure these points are clearly visible as in figure 1.

In [12]:

```
number_of_points = 10
normalise_sides_fig = plt.figure(figsize=(30,20))
plt.axis("off")
plt.title("Contours")
c = 1
for img_num in images_to_plot:
    rgb_img = imgs[img_num].copy()
    contours = get_puzzle_contour(msks[img_num])
    clockwise_contours = get_clockwise_contour(contours)
    side_contours = extract_sides(clockwise_contours, original_pixels[img_num])
    corners = np.int32(original_pixels[img_num]).reshape((-1,1,2))
    interpolated_0 = even_spaced_contour(side_contours[0], number_of_points)
    interpolated_1 = even_spaced_contour(side_contours[1], number_of_points)
    interpolated_2 = even_spaced_contour(side_contours[2], number_of_points)
    interpolated_3 = even_spaced_contour(side_contours[3], number_of_points)

    cv2.polylines(rgb_img, side_contours[0], False, colours[0] , 3)
    cv2.polylines(rgb_img, side_contours[1], False, colours[1], 3)
    cv2.polylines(rgb_img, side_contours[2], False, colours[2], 3)
    cv2.polylines(rgb_img, side_contours[3], False, colours[3], 3)
    cv2.circle(rgb_img, side_contours[0][0][-1], 15, colours[0], -1)
    cv2.circle(rgb_img, side_contours[1][0][-1], 15, colours[1], -1)
    cv2.circle(rgb_img, side_contours[2][0][-1], 15, colours[2], -1)
    cv2.circle(rgb_img, side_contours[3][0][-1], 15, colours[3], -1)
    for i in range(number_of_points):
        cv2.circle(rgb_img, interpolated_0[i], 10, white, -1)
        cv2.circle(rgb_img, interpolated_1[i], 10, white, -1)
        cv2.circle(rgb_img, interpolated_2[i], 10, white, -1)
        cv2.circle(rgb_img, interpolated_3[i], 10, white, -1)

    normalise_sides_fig.add_subplot(2, 3, c)
    plt.title(f"Image Number {img_num}")
    plt.axis("off")
    plt.imshow(rgb_img)

    normalise_sides_fig.add_subplot(2, 3, c+3)
    plt.axis('on')
    side_0 = transform_puzzle_side(side_contours[0])
    side_1 = transform_puzzle_side(side_contours[1])
    side_2 = transform_puzzle_side(side_contours[2])
    side_3 = transform_puzzle_side(side_contours[3])
    interpolated_a_side_0 = even_spaced_contour(side_0, number_of_points)
    interpolated_a_side_1 = even_spaced_contour(side_1, number_of_points)
    interpolated_a_side_2 = even_spaced_contour(side_2, number_of_points)
    interpolated_a_side_3 = even_spaced_contour(side_3, number_of_points)
    plt.plot(side_0[:,0], side_0[:,1], color= "red")
    plt.plot(side_1[:,0], side_1[:,1], color= "yellow")
    plt.plot(side_2[:,0], side_2[:,1], color= "blue")
    plt.plot(side_3[:,0], side_3[:,1], color= "green")
    plt.xlim([-1, 1])
    plt.ylim([-1, 1])
    plt.title(f"Image Number {img_num}")
    plt.axis("off")
    c+=1

plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
```

```
r floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] fo
r floats or [0..255] for integers).
```




# 3) Match Shape Models

## 3.1)

Before we can apply these rules, we need to know which sides of pieces are flat so we can determine if they should be matched or not, and so that we can determine if a piece is an interior piece (no flat sides), edge piece (1 flat side), corner piece (2 flat sides) or invalid piece (3 or 4 flat sides). Note that with side information, you could solve for possible the widths and heights of the puzzle in terms of pieces. Taking into account the number of pieces along the perimeter and the total area. Code is_flat_side(contour, min_ratio=0.9) that checks if the distance cont_dist between the two endpoints of a non-closed contour is approximately equal to the entire length cont_len of the c

In [13]:

```
def is_flat_side(contour, min_ratio=0.9):
    cont_dist = np.linalg.norm(contour[0] - contour[-1],axis=-1)
    cont_len = cv2.arcLength(contour,False)

    return (cont_dist/cont_len) >= min_ratio
```

## 3.2)

Using the function from the previous step construct an array of all the non-flat normalised sides that can be matched together, thus allowing us to satisfy (a). Construct another array that corresponds to this one that contains the tuples with the index of the puzzle piece and the index of the side in that puzzle piece, so that if we have an index from this new array we can map back to the original puzzle piece and side. This will be useful after obtaining the output from k-nearest neighbours.

In [14]:

```
number_of_points = 64

num_images = 48

non_flat_sides = []
non_flat_sides_indices =[]

flat_sides = []
flat_sides_indices = []

for i in range(num_images):
    image_sides = extract_sides(get_puzzle_contour(msks[i]), original_pixels[i])

    for side in image_sides:
        interpolated_side = even_spaced_contour(side)
        normalized_side = transform_puzzle_side(interpolated_side)

        if (is_flat_side(side[0])):
            flat_sides_indices.append(i)
        else:
            non_flat_sides.append(normalized_side)
            non_flat_sides_indices.append(i)

non_flat_sides = np.array(non_flat_sides)
non_flat_sides_indices = np.array(non_flat_sides_indices)
num_non_flats = non_flat_sides.shape[0]

rotated_sides = (non_flat_sides * (-1, -1))[:, ::-1, :]

non_flat_sides_features = non_flat_sides.reshape((num_non_flats,number_of_points * 2))
rotated_sides_features = rotated_sides.reshape((num_non_flats,number_of_points * 2))
```

## 3.3)

Remember in question 2.2.1 how we did not rotate the normalised sides if they were not protruding. We can easily satisfy rule (c) if we produce a training dataset from the array of normalised sides in the previous step except rotating by 180◦ and reversing the order of points in those sides. Since the sides are normalised, we can rotate by multiplying the x and y coordinates of all the points in the contours by −1, which will convert protruding sides to sunken sides and vice-versa. However, remember how the first point of the side contour should point to the left down the −ve x-axis, we have just violated this property by performing the rotation, thus we need to reverse the order of our rotated sides so they correspond once again. If we now find the k-nearest neighbours from the original non-rotated normalised sides, we minimise the distance between sunken and protruding points and maximise the distance if they are incompatible, see figure 2. Use sklearn to perform k-nearest neighbours, don't forget to flatten your sides into feature vectors, an incomplete example is given below:

In [15]:

```
# This code was given
knn = NearestNeighbors(n_neighbors=1, algorithm="brute")
knn = knn.fit(rotated_sides_features)
distances, indices = knn.kneighbors(non_flat_sides_features)
```

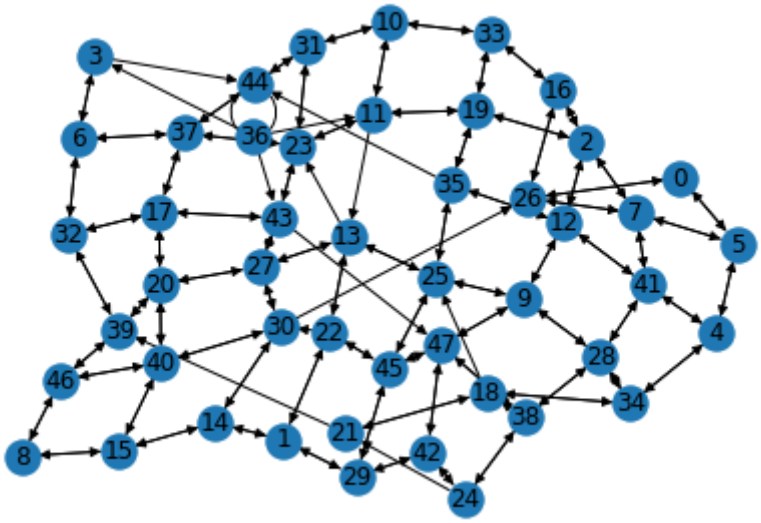
## 3.4)

In [16]:

```
# This code was given
def plot_graph(V, E, seed=42):
    random.seed(seed) # graphs are randomly plotted
    np.random.seed(seed) # graphs are randomly plotted
    G = nx.DiGraph()
    G.add_nodes_from(V)
    G.add_edges_from(E)
    nx.draw_kamada_kawai(G, with_labels=True)
    plt.show()
```

In [17]:

```
vertices = np.arange(48)
edges = np.zeros((non_flat_sides.shape[0],2),dtype=int)

for i in range(non_flat_sides.shape[0]):
    edges[i][0]=int(non_flat_sides_indices[i])
    edges[i][1]=int(non_flat_sides_indices[indices[i]])

plot_graph(vertices, edges)
```

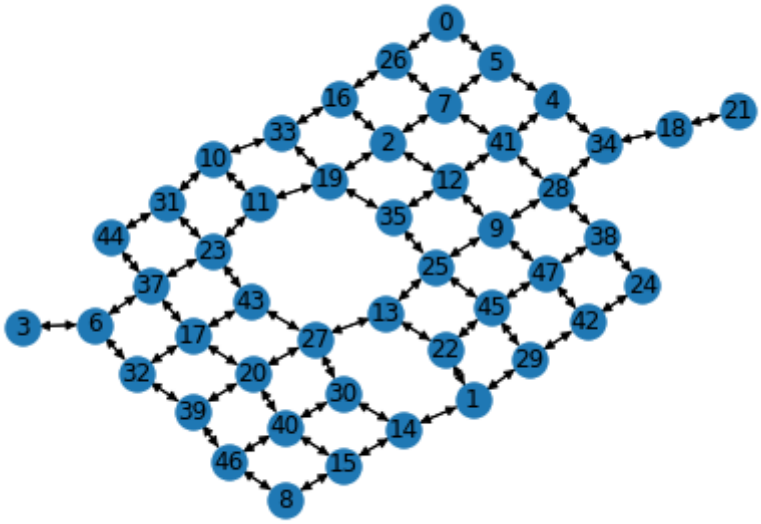



## 3.5)

Remember the edges are directed in the previous question, so if side a matches to side b, side b might not match back to side a. Plot a second graph that only includes edges where sides match back to themselves

In [18]:

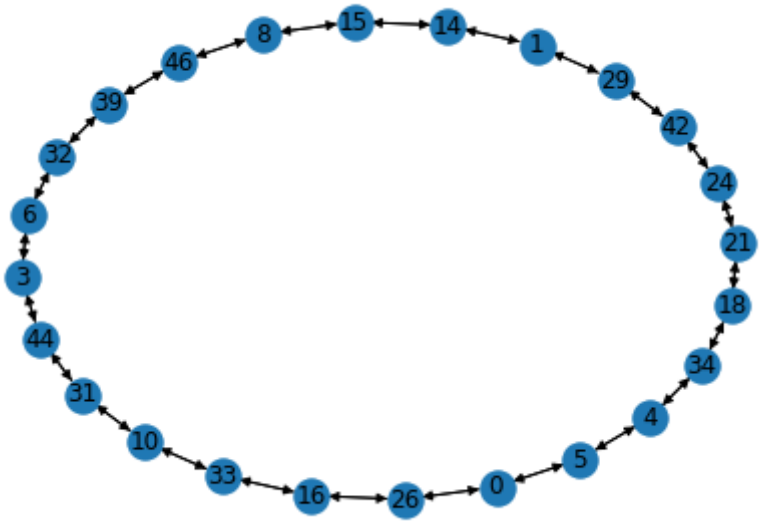
```
vertices2 = []
edges2 = []
for i in range(num_non_flats):
    for j in range(num_non_flats):
        if i !=j:
            if edges[i][0] == edges[j][1] and edges[i][1] == edges[j][0]:
                edges2.append(edges[i])
                vertices2.append(edges[i][0])
                vertices2.append(edges[i][1])

edges2 = np.array(edges2)
plot_graph(vertices2, edges2)
```




## 3.6)

In [21]:

```
vertices3 = []
edges3= [(24,21),(3,44),(21,24),(44,3)]
for i in range(edges2.shape[0]):
    if edges2[i][0] in flat_sides_indices and edges2[i][1] in flat_sides_indices:
        edges3.append(edges2[i])
        vertices3.append(edges2[i][0])
        vertices3.append(edges2[i][1])
plot_graph(vertices3, edges3)
```




## 3.7)

- The orientation of the side contours needs to be the same for all the steps.
- If the results of the corner detection algorithm are stored in a clockwise direction, but the contours are stored in an counterclockwise direction, we could have a situation where protruding edges and sunken edges are incorrectly labeled due to altered directions of the contours.

## 3.8)

- The pipeline managed to solve the puzzle relatively quickly.
- As the puzzle scales, the algorithm will take longer to solve the puzzle, but given a correct set of labels etc the algorithm should perform just as well.
- The issue with larger puzzle pieces will be the unreliable labels generated. This was a relatively small puzzle and in Lab 5 we saw the bias amongst different raters. With larger puzzles, this will only increase and affect the algorithm even more.