**COMPUTER VISION**

**CS – GY 6643**

**PROJECT: SNAKE GAME**

**TEAM:**

**HARSH JALUTHARIA – *hj2607***

**MUDIT NIGAM – *mn3439***

**RHUTHVIK DENDUKURI – *rd3377***

Introduction

# Background

The original Snake game, popularized by Nokia's monochrome mobile phones in the late 1990s, is a classic arcade-style video game. The objective is to control a continuously growing snake by guiding it to eat food items while avoiding collisions with its own body or the boundaries. As the snake grows longer with each piece of food consumed, the challenge lies in navigating through an increasingly crowded and confined space.

The game's simplicity and portability contributed to its enduring popularity, making it an ideal testbed for computer vision projects focused on tracking movement, detecting collisions, and developing advanced features like obstacle avoidance or path planning algorithms.

# Motivation

A great chance to create and test computer vision algorithms in a controlled setting is the Snake game. It presents various non-trivial issues for computer vision systems despite having straightforward gameplay mechanisms. Robust vision algorithms are needed to identify food items, follow the snake's movement accurately, and detect impacts with objects or the snake's own body. Furthermore, the continuous nature of the game and its rising complexity as the snake lengthens can be utilized to assess how well vision techniques operate and how scalable they are under different circumstances. Starting with this well-defined and well-known game allows researchers to concentrate on creating unique vision techniques without being distracted by intricate rendering or physics models.

# Objectives

## Implement a Snake Game

The primary objective is to develop the core mechanics of the snake game, encompassing essential elements such as snake movement, food and hurdle generation and a scoring system. By recreating the classic gameplay, we aim to provide a familiar yet enjoyable experience for players.

## Integrate Computer Vision

Leveraging computer vision algorithms, we seek to detect objects within the game environment and enable players to control the snake’s movements using real-world objects captured by the system’s web camera. This integration adds a unique layer of interactivity to the game, allowing players to engage with the virtual world in innovative ways.

## Enhance Gameplay Experience

The goal is to create an engaging and intuitive gameplay experience by seamlessly blending computer vision technology with the classic game mechanics.

## Scalability and Performance

It is crucial to design the system to be scalable and efficient, capable of running in real-time in standard hardware while delivering a smooth experience. By optimizing algorithms and leveraging hardware acceleration where possible, we aim to ensure that the game performs reliably across a range of devices and environments.

Project Overview

# Description

This project reimagines the Snake Game by integrating computer vision. Players control the snake, collecting food and avoiding obstacles. What sets it apart is the ability to control the snake using real-world objects detected through a camera feed. Computer vision algorithms analyze the feed in real-time, translating object positions into commands for the snake's movement. This immersive gameplay experience blends reality with virtuality, offering dynamic interaction. Whether navigating household items or custom obstacles, players enjoy a fusion of nostalgia and innovation.

# Technologies Used

1. Python
2. OpenCV
3. Numpy
4. Tkinter
5. Random Module

# CV techniques Used

## Color – based Segmentation

This involves segmenting the camera feed based on color to isolate the objects of interest from the scene. This is typically done using color thresholding techniques, where pixels within a certain color range are considered part of the object, while the others are discarded. In our scenario, this process is implemented using the range from the color codes “(29, 86, 18)” (lowest green HSV code) to “(93, 255, 255)” (highest green HSV code).

## Preprocessing

Before further analysis, the segmented image may undergo preprocessing steps like erosion and dilation, which are further discussed, to enhance the object boundaries and remove noise. These operations helps refine the object masks and improve accuracy of subsequent processing steps.

## Contour detection

Once the preprocessed image is computed, contours are detected using “cv2.findContours()” function. Contours represent the boundaries of connected regions within the segmented image and are essential for identifying and analyzing objects.

The input parameters of this function take an image, contour retrieval mode and contour approximation method. For our necessity, the contour retrieval mode is set to cv2.RETR\_EXTERNAL which retrieves only the external contours (outer boundary of the object) and the contour approximation method is set to cv2\_CHAIN\_APPROX\_SIMPLE which compresses horizontal, vertical and diagonal segments leaving only their endpoints.

Another way to find the contours is y Using Hough Transform but it had several complexities.

Using Hough Transform with edge orientation was a challenge we faced in this project. This method involves detecting edges and computing edge orientations before applying hough transform. This is a complex computational approach as the frames give a continuous input to the system to which hough transform has to be applied and the result has to be provided within no time. The accuracy of edge detection affects the performance of hough transform.

In this scenario, the factors like light conditions, object textures and noise levels play a strong role in quality of edge detection. In addition, edge detection is sensitive to noise in the image. Though a gaussian blur is used to smoothen the image and lessen the noise, the computed edges were not sufficiently clear for the hough transform to perform with accuracy. Also, performing edge detection and orientation estimation for every frame in such a real – time scenario imposes significant computational overhead.

In conclusion, since such edge detection and orientation need algorithms with strict time constraints, “cv2.findContours” performed both accurately and faster.

## Object Localization and Identification

Object localization and identification are critical components that enable precise interaction between the virtual snake and real-world items. Through meticulous contour analysis, the system extracts vital geometric information from observed objects, including their sizes, shapes, and spatial orientations. This analysis ensures the accurate localization and tracking of objects within the gaming environment, facilitating realistic interaction with the virtual snake.

Leveraging the derived contour information, the game dynamically responds to the player's actions, offering seamless and captivating gameplay experiences. By applying additional criteria to filter out extraneous contours and identify objects of interest, the system distinguishes between valid objects and noise or background elements. This process enhances gameplay dynamics by ensuring accurate interaction between the virtual snake and detected objects, delivering a more immersive and responsive gaming experience for players.

# Implementation Details

## Game Loop/ Logic

### Initialization:

Certain variables have been initialized to establish the initial state of the game. Namely:

* + - Snake Length: initial length of the snake at the start of the game.
    - Score: initial score of the game is set to 0
    - Number of Hurdles: Initializing the number of hurdles to appear per game on the game frame.
    - Win Score: Acts as a win criterion. The score to be achieved to win the game.
    - Food Image: Assigning an image to the food in the game, the image file path is stored.
    - Hurdle Image: Assigning an image to the hurdle in the game, the image file path is stored.

### Main Loop:

* + Continuous Frame Capture: The main loop continuously captures frames from the camera feed to provide real-time input for the game environment.
  + Color – Based Segmentation: Color – based segmentation is applied to each captured frame to isolate objects of interest, particularly snake’s head. This involves pixels within a specified color range that likely belongs to the snake’s head.
  + Contour Detection: Contours representing potential positions of the snake's head are detected within the segmented image. These contours outline regions where the snake's head may be located, enabling precise localization.
  + Snake’s head Identification: Among the detected contours, the contour with the maximum area is identified as the snake's head. This contour likely corresponds to the largest object within the image, which is assumed to be the snake's head.
  + Movement Direction Update: The movement direction of the snake is updated based on the difference between the current and previous positions of its head. This ensures that the snake moves smoothly and responsively in the desired direction.
  + Collision detection: Collision detection is performed to check for collisions between the snake's head and obstacles or its own body. This prevents the snake from intersecting with obstacles or itself, which would result in game over.
  + Game Over/ Win Condition Check: If the snake collides with an obstacle or its own body, the game ends, and the player is prompted to restart. Additionally, the game checks for win conditions, where the player reaches a predefined score threshold, leading to victory.
  + Score and Snake Length Update: If the snake consumes food, its length increases, and the player's score increments. This rewards the player for successfully navigating the snake to consume food items.
  + Display Update: The game continuously updates the display with the current frame, snake, food, obstacles, and score. This ensures that the player receives real-time feedback on the game state and their progress.
  + Loop Continuation: The main loop continues iterating until the player chooses to exit the game. This allows for uninterrupted gameplay and provides the player with the option to play as long as desired.

### Reset Functionality:

Upon restarting the game, all variables are reset to their initial states, and new obstacle and food locations are generated.

## Code Structure

### Main Script

* + The main script initializes the game frame, including setting parameters like WIN\_SCORE, MAX\_OBSTACLES, and video dimensions (VIDEO\_WIDTH, VIDEO\_HEIGHT).
  + It imports necessary libraries such as NumPy, OpenCV, Tkinter
  + Global variables are defined, including “playGame” to control the game loop.

### SnakeGame Class

* + The SnakeGame class encapsulates the core game logic and functionalities.
  + It manages the snake's movement, food generation, obstacle placement, collision detection, and score tracking.
  + Methods within this class handle game initialization, resetting, updating snake movement, detecting collisions, and determining win conditions.
  + The class also includes functions for generating obstacles, randomizing food locations, drawing game elements on the screen, and handling game over/win conditions.

### Computer Vision Techniques

* + The code utilizes computer vision techniques such as color-based segmentation, contour detection, and morphological operations for image processing.
  + Functions for eroding and dilating image masks, as well as finding contours and analyzing their properties, are implemented to extract relevant information from the camera feed.

### Contour Analysis and Object Detection

* + Contour analysis plays a crucial role in extracting geometric information from observed objects, enabling precise localization and tracking within the game environment.
  + Object identification involves filtering and analyzing contours to distinguish between valid objects (e.g., food items or obstacles) and noise or background elements.
  + Hough Transform with edge orientation, although not explicitly implemented in the provided code, could potentially be utilized for object detection, albeit with challenges related to complexity and computational overhead.

### Game Loop and Logic

* + The game loop orchestrates the entire gameplay experience, continuously capturing frames from the camera feed and updating the game state accordingly.
  + It handles snake movement, collision detection, score tracking, and user interaction within the main loop.
  + Upon game over or win conditions, appropriate messages are displayed, and the player is given the option to restart the game or exit the application.

### Reset Functionality and Game Over/Win Condition Handling:

* + Functions are implemented to handle game reset, ensuring that all variables are reinitialized, and new obstacle and food locations are generated.
  + Messages and prompts are displayed to the player upon game over or win conditions, allowing them to choose whether to continue playing or exit the game.

This structured code organization enhances readability, maintainability and extensibility, enabling easier implementation of new features and improvements.

# Game and General Implementations

## UI Design

The UI design of the game is implemented usin gTkinter, a python library to create graphical user interaces.

### Camera Feed Display

The UI prominently features a display area showing the real-time camera feed captured by the device's camera. This feed serves as the backdrop for the game environment, providing visual context for the player's interactions.

### Snake Representation

Within the camera feed display, the snake is represented by a continuous line segment or series of connected points, denoting its current position and trajectory. The snake's appearance may vary in color or thickness to enhance visibility and distinguish it from other elements in the environment.

### Food and Hurdle Visualization

Food items and obstacles are visually represented within the camera feed display. Food items appear as distinct objects or icons that the snake must consume to grow and earn points. Obstacles are depicted as static elements or barriers that the snake must navigate around to avoid collision.

### Score Display

A dedicated area of the UI is allocated for displaying the player's current score. This numerical value updates dynamically as the player successfully consumes food items and earns points throughout the gameplay session.

### Game Over/Win Condition Message

Tkinter's message box or label widgets are utilized to display relevant messages on the UI in the event of game over or achievement of win conditions. Tkinter's message box module may be employed to prompt the player with options to restart the game or exit the application.

### User Interface Prompts

Interactive buttons or widgets created using Tkinter are incorporated into the UI to facilitate user interaction. Tkinter's button widget can be configured to trigger actions such as restarting the game or exiting the application in response to player input.

### Visual Feedback and Animations

The UI may include visual feedback elements or animations to enhance the user’s engagement. Tkinter’s canvas widget supports animation capabilities, allowing for the creation of dynamic feedback elements.

## Snake Movement Mechanism

The snake’s movement is controlled by tracking a green object (presumably a green ball or marker) using color thresholding and contour detection techniques from the OpenCV library.

* + The code captures frames from the webcam using “cv2.VideoCapture”.
  + It applies color thresholding to detect green objects in the frame. This is done by converting the frame to the HSV color space and creating a binary mask based on the green color range defined by GREEN\_LOWER\_THRESHOLD and GREEN\_UPPER\_THRESHOLD.
  + The mask is further refined using morphological operations (erosion and dilation) to remove noise and fill small holes.
  + The code finds contours in the binary mask using “cv2.findContours”. The largest contour is assumed to be the green object used for controlling the snake.
  + The contour is converted to a list of “Point” objects (a custom class defined in the code).
  + Welzl's algorithm is applied to the list of points to find the minimum enclosing circle. This is essentially finding the smallest circle that contains all the points in the contour.
  + The center and radius of the minimum enclosing circle are obtained.
  + If the radius is greater than a certain threshold (10 in this case), the center of the circle is considered as the new head position of the snake, and the “updateSnake” function is called to update the snake's position and check for collisions, food consumption, and game over conditions.
  + If the radius is smaller than the threshold, it means the green object is too small or not detected properly, so the snake's position is not updated, but the existing snake and other game objects are still drawn on the frame.
  + Transitions between consecutive positions of the snake are executed smoothly to ensure seamless movement within the game environment. This is achieved by updating the snake's position gradually, creating fluid motion and enhancing the overall visual appeal of the game.

The snake's movement is controlled by tracking a green object using computer vision techniques. The center of the minimum enclosing circle around the green object's contour is used as the new head position of the snake, and the snake's body is updated accordingly while checking for various game conditions.

## Food and Hurdle Generation

### Food Generation (Apple)

* + - * The “generateRandomFoodLocation” method is called to generate a new food location whenever the snake eats the existing food item.
      * The method generates random x and y coordinates within the game screen boundaries (between 50 and “VIDEO\_WIDTH-50” for x, and between 50 and “VIDEO\_HEIGHT-50” for y). This ensures that the food item is not placed too close to the edges of the screen.
      * The method then checks if the randomly generated food location is at least “MIN\_DISTANCE\_FOOD\_OBSTACLE” distance away from all the existing obstacle locations (“self.obstacleLocations”). This distance is set to 60 pixels by default.
      * If the generated food location is too close to any obstacle, the method discards that location and generates a new random location.
      * This process continues until a suitable location is found that satisfies the minimum distance constraint from all obstacles.
      * Once a valid location is found, it is stored in “self.foodLocation”, and the corresponding food image (“self.foodImage”) is drawn at that location using the “drawImageOverFrame” function.
      * The “drawImageOverFrame” function takes the game frame, the food image, and the location as input, and it draws the food image centered at the specified location by overlaying it on the frame.

### Hurdle Generation (Building)

* The “generateObstacles” method is called to generate the obstacle locations during the game initialization.
* The method first initializes an empty list “self.obstacleLocations” to store the obstacle locations.
* It then enters a loop that continues until the number of obstacles in “self.obstacleLocations” reaches “self.obstacleCount”, which is set to “MAX\_OBSTACLES” (5 by default).
* Inside the loop, a random location (“temp\_point”) is generated within the game screen boundaries (between 50 and “VIDEO\_WIDTH-50” for x, and between 50 and “VIDEO\_HEIGHT-50” for y). This ensures that the obstacles are not placed too close to the edges of the screen.
* The method then checks if the randomly generated “temp\_point” is at least “MIN\_DISTANCE\_BW\_OBSTACLES” distance away from all the existing obstacle locations in “self.obstacleLocations”. This distance is set to 60 pixels by default.
* If the generated “temp\_point” is too close to any existing obstacle, the method discards that location and generates a new random location.
* This process continues until a valid location is found that satisfies the minimum distance constraint from all other obstacles.
* Once a valid location is found, it is added to “self.obstacleLocations”.
* After the loop completes, “self.obstacleLocations” contains the locations of all the obstacles.
* The obstacles are drawn on the game screen using the “drawObjects” method, which iterates over “self.obstacleLocations” and draws the obstacle image (“self.obstacleImage”) at each location using the “drawImageOverFrame” function, similar to how the food image is drawn.

The minimum distance constraints (“MIN\_DISTANCE\_FOOD\_OBSTACLE” and “MIN\_DISTANCE\_BW\_OBSTACLES”) ensure that the food item and obstacles are not placed too close to each other or to the edges of the screen, providing a fair and playable game environment.

## Collision Detection

### Collision with Hurdles

* + - For each obstacle location in “self.obstacleLocations”, the method calls the “checkPointOverlapImage” function, passing the new head location (“NewHeadLocation”), the obstacle location, and the obstacle image size (“self.obstacleImageSize”).
    - The “checkPointOverlapImage” function calculates the bounding box around the obstacle image based on its location and size. It checks if the new head location falls within this bounding box.
    - If the new head location overlaps with any obstacle's bounding box, it means a collision has occurred.
    - In case of a collision, the “gameFinished” method is called with “win=False”, indicating a game over condition.
    - The snake's body is not a perfect circle or rectangle, making it challenging to detect collisions accurately. We had to use the “cv2.polylines” and “cv2.pointPolygonTest” functions to handle this irregularity.

### Collision with Snake’s Body

* The method creates a numpy array “check\_pts” containing all the points of the snake's body except the last few points (excluding the head and a few segments near the head).
* The “cv2.polylines” function is used to draw these points as a polygon on the game frame.
* The “cv2.pointPolygonTest” function is then used to calculate the minimum distance between the new head location (“NewHeadLocation”) and the polygon formed by the snake's body points.
* If the minimum distance is between -1 and 1 (inclusive), it means the new head location is within or on the boundary of the polygon, indicating a collision with the snake's body.
* In case of a collision, the “gameFinished” method is called with “win=False”, indicating a game over condition.
* Checking collisions between the snake's head and every point in its body can be computationally expensive, especially as the snake grows longer. We had to optimize the process by excluding the few points near the head from the collision check, as they are unlikely to collide with the head.

There were edge cases to consider, such as when the snake's head is exactly on the boundary of its body or an obstacle. We had to handle these cases carefully to ensure accurate collision detection. Collision detection had to be integrated with the computer vision components of the game, which added complexity. The new head location was obtained from the minimum enclosing circle around the green object's contour, and this had to be correctly used for collision checks.

Implementing collision detection required careful consideration of geometric calculations, edge cases, performance optimization, and integration with the computer vision components of the game.

# Computer Vision Technique Implementations

## Blur

Gaussian blur is a common image processing technique used to reduce noise and smooth out images by averaging the pixel values in a local neighborhood around each pixel. In this project, the blur operation is applied using a Gaussian kernel with a size of 11x11 pixels and a standard deviation of 2.

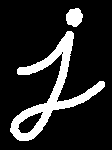
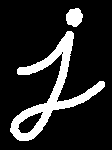
By blurring the image, high-frequency noise is attenuated, and small-scale details are smoothed out, resulting in a more uniform appearance. This can be beneficial for subsequent processing steps, such as edge detection or object recognition, as it helps to enhance the clarity of important features while reducing the impact of irrelevant or distracting elements in the image. Overall, Gaussian blur is a versatile tool commonly used in various computer vision applications to improve the quality and interpretability of images.

## Mask

The mask is a binary image derived from an image using thresholding and logical operations. It isolates regions in the image where the pixel values correspond to the specified green color range defined by GREEN\_LOWER\_THRESHOLD and GREEN\_UPPER\_THRESHOLD. White pixels in the mask indicate the presence of green colors within the specified range, while black pixels represent non-green areas. This mask is crucial for identifying and segmenting objects or features of interest in the image, facilitating subsequent processing steps such as object detection or tracking.

## Erode

The function erode performs erosion on a binary mask. Erosion is a morphological operation that shrinks the boundaries of foreground objects in an image. It is commonly used to remove small objects, smooth contours, or separate overlapping objects.



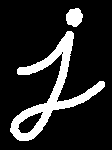
Here's how the function works:

* Inputs:
  + mask: A binary image where foreground objects are represented by white pixels (255) and background by black pixels (0).
  + kernel\_size: A tuple specifying the size of the kernel used for erosion. Kernel of odd size (3, 5, 7). In this project, it's a 3x3 square kernel.
  + iterations: The number of times erosion is applied. In this project, we have set it to 1.
* Initialization:
  + The function initializes a square structuring element (kernel) with the specifiedsize using NumPy's np.ones() function. The structuring element is a binary matrix that defines the neighborhood used for the erosion operation.
* Erosion Operation:
  + The erosion operation is applied iterations times to the input mask. For each iteration, the function performs the following steps:
    - It creates a copy of the input mask called eroded\_mask.
    - It loops through each pixel in the mask.
    - At each pixel, it computes the minimum value of the neighbourhood defined by the kernel.
    - The minimum value represents the erosion effect, where the centre pixel of the kernel is set to the minimum value of the pixels within its neighbourhood.
    - This process effectively shrinks the boundaries of the foreground objects in the mask.
* Return Value:
  + After applying erosion for the specified number of iterations, the function returns the resulting eroded mask.

Overall, the “erode()” function provides a way to perform erosion on binary masks, which can be useful for various image processing tasks, such as noise reduction, segmentation, and shape analysis. Adjusting the kernel size and number of iterations allows for control over the erosion effect and its extent.

## Dilate

The function dilate(mask, iterations=2) is the sister function of erosion and performs dilation on a binary mask. Dilation is a morphological operation that expands the boundaries of foreground objects in an image. It is commonly used to fill gaps, join fragmented objects, or increase the size of objects.



Here's how the function works:

* Inputs:
  + mask: A binary image where foreground objects are represented by white pixels (255) and background by black pixels (0).
  + iterations: The number of times dilation is applied. In this project, we have set it to 2.
* Dilation Operation:

The dilation operation is applied iterations times to the input mask. For each iteration, the function performs the following steps:

* It creates a copy of the input mask called dilated\_mask.
* It loops through each pixel in the mask.
* At each pixel, it computes the maximum value of the neighborhood defined by a structuring element.
* The maximum value represents the dilation effect, where the center pixel of the structuring element is set to the maximum value of the pixels within its neighborhood.
* This process effectively expands the boundaries of the foreground objects in the mask.
* Return Value:
  + After applying dilation for the specified number of iterations, the function returns the resulting dilated mask.

Overall, the dilate() function provides a way to perform dilation on binary masks, which can be useful for various image processing tasks, such as filling gaps, connecting components, and enhancing features. Adjusting the number of iterations allows for control over the dilation effect and its extent.

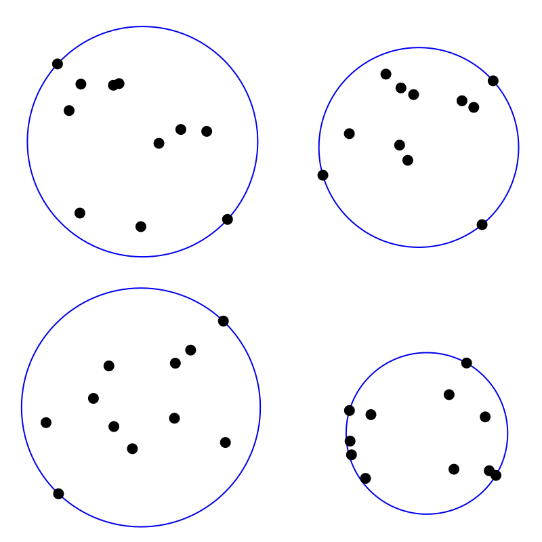
## Find Contour

“findContours” is used to identify regions of interest within an image. First, the function detects contours in a binary image mask, which typically results from thresholding or other segmentation techniques. After finding the contours, the function to find line contours is executed. This step is necessary for compatibility and consistency across different versions of OpenCV. The grab\_contours function reshapes the contours into a standardized format suitable for further processing. This ensures that the contours can be easily manipulated and analyzed in subsequent steps of the image processing pipeline.

Overall, these lines of code play a crucial role in identifying and extracting regions of interest from the input image, facilitating tasks such as object detection, shape analysis, and image segmentation in various computer vision applications.

## Find MEC (Welzl’s Algorithm)

The Welzl function implements the Welzl's algorithm, which recursively computes the minimum enclosing circle for a set of points. It takes a list of 2D points as input and returns the minimum enclosing circle. The Recursive algorithm that can find the minimal circle in O(n) time for a set of n points.



Here's how the function works:

* Inputs:

The function takes a list of 2D points (P) as input. These points represent the set of points for which the minimum enclosing circle needs to be found.

* Initialization: The function initializes some helper functions and classes:
  + Point: Represents a 2D point with X and Y coordinates.
  + Circle: Represents a 2D circle with a center (C) and a radius (R).
  + dist(a, b): Calculates the Euclidean distance between two points.
  + is\_inside(c, p): Checks if a point lies inside or on the boundaries of a circle.
  + get\_circle\_center(bx, by, cx, cy): Helper method to compute the center of a circle given three points. For 3 points, the circle is the circumcircle of the triangle formed by the points.
  + def circle\_from1(A: Point, B: Point): Computes the MEC intersecting two points by setting the center at their midpoint and radius as half the distance
  + def circle\_from2(A: Point, B: Point, C: Point): Computes a unique MEC intersecting three points using geometric calculations for the center and radius.
  + def is\_valid\_circle(c: Circle, P: list[Point]): Checks if the given circle encloses all points in the list by verifying each point's position relative to the circle.
  + def min\_circle\_trivial(P: list[Point]): Determines the MEC for a small number of points (0, 1, 2, or 3) directly, handling base cases.
* Working:
  + The main function welzl() is a wrapper function that shuffles the input points randomly and then calls the helper function welzl\_helper() with the shuffled points, an empty list (representing the set of points on the circle boundary), and the number of points.
  + The welzl\_helper() function is a recursive function that computes the minimum enclosing circle.
  + At each recursive call, it selects a random point from the input points.
  + It then recursively computes the minimum enclosing circle for the remaining points without the selected point.
  + If the selected point lies inside the computed circle, the circle is returned.
  + Otherwise, the selected point is added to the boundary set, and the minimum enclosing circle is computed recursively with the updated sets of points.
  + The recursion terminates when either all points are processed or the boundary set contains three points (which determines the circle uniquely).
* Return Value:
  + The function returns the minimum enclosing circle that encloses all the input points.

Overall, the Welzl function efficiently computes the minimum enclosing circle for a set of 2D points using the Welzl's algorithm, which has a time complexity of O(n) on average, where n is the number of input points.