1 Question 1

For this question, I suppose we can't use any mature SQLs, and need to design customized tables storing the point and polygon data. The SQL-like *create-table* languages are as follows.

```sql
-- Table for Flora and Fauna Sightings
-- Stores individual point sightings with time and species information.
CREATE TABLE Sighting (
    sighting_id INT PRIMARY KEY AUTO_INCREMENT, -- A unique identifier for each sighting
    species_name VARCHAR(255) NOT NULL,         -- The name of the species observed
    sighting_time DATETIME NOT NULL,            -- The date and time of the sighting
    latitude DECIMAL(9, 6) NOT NULL,            -- Latitude of the sighting location (e.g., -90.000000 to +90.000000)
    longitude DECIMAL(9, 6) NOT NULL            -- Longitude of the sighting location (e.g., -180.000000 to +180.000000)
);

-- Table for Wetlands
-- Stores information about each defined wetland area.
CREATE TABLE Wetland (
    wetland_id INT PRIMARY KEY AUTO_INCREMENT,  -- Unique identifier for each wetland
    wetland_name VARCHAR(255) NOT NULL UNIQUE,  -- Name of the wetland (unique for identification)
    description TEXT                            -- Optional: A longer description of the wetland
);
-- Table to store the vertices (points) that make up each wetland polygon.
-- Each row represents one point along the boundary of a specific wetland.
CREATE TABLE WetlandVertex (
    wetland_id INT NOT NULL,                    -- Foreign key linking to the Wetland table
    sequence_number INT NOT NULL,               -- The order of this vertex in the polygon (e.g., 1, 2, 3...)
    latitude DECIMAL(9, 6) NOT NULL,            -- Latitude of the vertex
    longitude DECIMAL(9, 6) NOT NULL,           -- Longitude of the vertex
    PRIMARY KEY (wetland_id, sequence_number),  -- Composite primary key to ensure unique sequence per wetland
    FOREIGN KEY (wetland_id) REFERENCES Wetland(wetland_id)
        ON DELETE CASCADE                       -- If a wetland is deleted, its vertices are also deleted
);

-- Table for State and National Forest Parks
-- Stores information about each defined forest park area.
CREATE TABLE ForestPark (
    park_id INT PRIMARY KEY AUTO_INCREMENT,     -- Unique identifier for each park
    park_name VARCHAR(255) NOT NULL UNIQUE,     -- Name of the forest park
    park_type VARCHAR(50) NOT NULL,             -- Type of park (e.g., 'State', 'National')
    description TEXT,                           -- Optional: A longer description of the park
    CHECK (park_type IN ('State', 'National'))  -- Constraint to ensure valid park types
);
-- Table to store the vertices (points) that make up each forest park polygon.
-- Each row represents one point along the boundary of a specific forest park.
CREATE TABLE ForestParkVertex (
    park_id INT NOT NULL,                     -- Foreign key linking to the ForestPark table
    sequence_number INT NOT NULL,             -- The order of this vertex in the polygon
    latitude DECIMAL(9, 6) NOT NULL,          -- Latitude of the vertex
    longitude DECIMAL(9, 6) NOT NULL,         -- Longitude of the vertex
    PRIMARY KEY (park_id, sequence_number),   -- Composite primary key
    FOREIGN KEY (park_id) REFERENCES ForestPark(park_id)
        ON DELETE CASCADE                     -- If a park is deleted, its vertices are also deleted
);
```

Listing 1: Database Build

2 Question 2

To achieve the target query instructions, we need to first design the following operations.

```
MAKE_POINT(latitude, longitude)                -- Constructs a geometric point object
MAKE_POLYGON(vertex_table_name, id_column, id_value)       -- Constructs a geometric
    polygon object by fectching and ordering vertices from the respective vertex table
ST_Contains(geometry1, geometry2)              -- Returns TRUE if geometry1 completely
    contains geometry 2
ST_Distance(geometry1, geometry2)              -- Returns the shortest distance between
     two geometries
```

Listing 2: Operation Definition

2.1 Question 2.1

Filter by `species_name` and then perform a `point-in-polygon` test to see if each sighting's location is within the assigned park's boundary.

```sql
SELECT COUNT(s.sighting_id)
FROM Sighting s
WHERE
    s.species_name = 'legless lizard'
    AND ST_Contains(
        MAKE_POLYGON('ForestParkVertex', 'park_id', (SELECT park_id FROM ForestPark
    WHERE park_name = 'Pine Ridge Conservation Park')),
        MAKE_POINT(s.latitude, s.longitude)
    );
```

2.2 Question 2.2

Construct polygon objects for both wetlands and specified forest parks, then perform a `Polygon-in-Polygon` test.

```sql
SELECT w.wetland_name
FROM Wetland w
WHERE EXISTS
(
    SELECT 1
    FROM ForestPark fp
    WHERE
        fp.park_type IN ('State', 'National')
        AND ST_Contains(
            MAKE_POLYGON('ForestParkVertex', 'park_id', fp.park_id),
            MAKE_POLYGON('WetlandVertex', 'wetland_id', w.wetland_id)
        )
);
```

2.3 Question 2.3

Filter by `species_name`, construct a point for each sighting and polygons for all wetlands, then find the minimum distance from a point to any of the polygons, with special handling for point-in-polygon to set distance to 0.

```sql
SELECT
    s.sighting_id,
    s.latitude,
    s.longitude,
    MIN(
        CASE
            WHEN ST_Contains(MAKE_POLYGON('WetlandVertex', 'wetland_id', w.wetland_id)
    , MAKE_POINT(s.latitude, s.longitude)) THEN 0.0
            ELSE ST_Distance(MAKE_POINT(s.latitude, s.longitude), MAKE_POLYGON('
    WetlandVertex', 'wetland_id', w.wetland_id))
        END
    ) AS distance_to_closest_wetland_km
```

```sql
FROM
    Sighting s,
    Wetland w -- Cross join is implied here to check each sighting against each
    wetland
WHERE s.species_name = 'platypus'
GROUP BY s.sighting_id, s.latitude, s.longitude;
```

3 Question 3

For the following three questions, I also visualized them using IPython, which can be found in this GitHub link.[1]

3.1 Question 3.1

Input: speciesName: Name of the target species
parkName: Name of the target park
Output: sightingCount: Total number of sightings of speciesName within parkName

```
// 1. Retrieve Sighting Points for Species
rawSightings ← DB_Sightings.GET_POINTS_WHERE(species = speciesName)
pointsForQuadtree ← MAP(rawSightings, λs : (s.ID, MAKE_POINT(s.Lon, s.Lat)))
// 2. Build Quadtree for Species Sightings
quadtree ← NewQuadtree(pointsForQuadtree's Global BBox)
for each (id, point) in pointsForQuadtree do
    quadtree.Insert(id, point)
end
// 3. Retrieve and Prepare Park Polygon
parkPolygon ← DB_ForestPark.GET_POLYGON_WHERE(name = parkName)
/* Minimized Data: Fetches only the vertices for the *single target park*.
   */
preparedParkPolygon ← OPTIMIZE_FOR_CONTAINS(parkPolygon)
 parkBBox ← GET_BOUNDING_BOX(parkPolygon)
// 4. Spatial Filter (Quadtree Query)
candidateSightings ← quadtree.QueryRange(parkBBox)
/* Minimized Spatial Operations: Uses the Quadtree's spatial index to perform
   a fast, approximate MBR (Bounding Box) filter. Only points whose MBRs
   intersect the park's MBR are selected for the next step. This drastically
   reduces the number of points for the expensive exact geometric test.     */
// 5. Spatial Refinement (Exact Point-in-Polygon Test)
finalSightingsInPark ← ∅
for each (sightingID, pointObject) in candidateSightings do
    if preparedParkPolygon.Contains(pointObject) then
        finalSightingsInPark.Add(sightingID)
    end
end
/* Minimized Spatial Operations: The expensive Contains operation is
   performed only on the *reduced set* of candidates identified by the
   Quadtree, not all original sightings.                                    */
// 6. Return Count
sightingCount ← |finalSightingsInPark|
return sightingCount
```

[1]The IPython visualizations for these concepts are available at https://github.com/gjskywalker/COMP6311G_Assignments/blob/main/Assignment_1.ipynb

3.2 Question3.2

Input: targetParkTypes: List of park types to consider (e.g., ["State", "National"])
Output: wetlandsInParks: List of names of wetlands that are entirely within any of the target parks

```
// 1. Retrieve all Wetland Polygons
allWetlands ← DB_Wetlands.GET_ALL_POLYGONS()
/* Minimized Data: All wetland geometries are fetched once, then stored
   in-memory for R-tree construction and future precise checks.          */
wetlandPolygons ← MAP(allWetlands, λw : (w.ID, MAKE_POLYGON(w.Vertices)))

// 2. Build R-tree for Wetland Polygon MBRs
RtreeIndex ← NewRtree()
for each (id, polygon) in wetlandPolygons do
    RtreeIndex.Insert(id, GET_BOUNDING_BOX(polygon))
end
/* Minimized Data: R-tree is built *in-memory* on the fetched wetland data,
   avoiding repeated DB access. It indexes MBRs, which are simpler than full
   polygons.                                                             */

// 3. Retrieve and Prepare Target Park Polygons
targetParks ← DB_ForestPark.GET_POLYGONS_WHERE(type ∈ targetParkTypes)
/* Minimized Data: Fetches only the geometries for parks of the specified
   types.                                                                */
preparedParkPolygons ← MAP(targetParks, λp :
 (p.ID, OPTIMIZE_FOR_CONTAINS(MAKE_POLYGON(p.Vertices)))

foundWetlandsIDs ← ∅
for each (parkID, parkPolygon) in preparedParkPolygons do
    // 4. Spatial Filter (R-tree Query for candidates)
    parkBBox ← GET_BOUNDING_BOX(parkPolygon)
    candidateWetlandIDs ← RtreeIndex.QueryRange(parkBBox)
    /* Minimized Spatial Operations: R-tree performs a fast MBR
       intersection/containment search. Only wetlands whose MBRs *could* be
       inside the park's MBR are considered, drastically pruning the search
       space.                                                            */
    // 5. Spatial Refinement (Exact Polygon-in-Polygon Test)
    for each wetlandID in candidateWetlandIDs do
        wetlandPolygon ← wetlandPolygons[wetlandID]
        if parkPolygon.Contains(wetlandPolygon) then
            foundWetlandsIDs.Add(wetlandID)
        end
    end
    /* Minimized Spatial Operations: The expensive Contains operation is
       performed only on the *reduced set* of candidates identified by the
       R-tree, not all wetlands.                                         */
end

// 6. Return Result
wetlandsInParks ← MAP(foundWetlandsIDs, λid : wetlandPolygons[id].Name)
return wetlandsInParks
```

3.3 Question3.3

```
Input: speciesName: Name of the target species ("platypus")
Output: results: List of records: (sighting_id, latitude, longitude, distanceToWetland)

// 1. Retrieve Platypus Sighting Locations
platypusSightings ← DB_Sightings.GET_POINTS_WHERE(species = speciesName)

// 2. Retrieve all Wetland Polygons
wetlandPolygons ← DB_Wetlands.GET_ALL_POLYGONS()
/* Fetches all vertex data for all wetlands.                                  */

// 3. Build an R-tree on Wetland Polygons
RTreeWetlands ← NewRTree()
for each (wetlandID, polygon) in wetlandPolygons do
    MBR ← GET_MINIMUM_BOUNDING_RECTANGLE(polygon)
    RTreeWetlands.Insert(wetlandID, MBR)
end
/* Constructs a R-tree using the MBRs of wetlands for efficient spatial
   indexing.                                                                  */

// 4. Iterate through each Platypus Sighting
results ← ∅
for each sighting in platypusSightings do
    sightingPoint ← MAKE_POINT(sighting.Lon, sighting.Lat)
    minDistance ← ∞
    foundInside ← false

    // a. Find closest wetland candidates (R-tree K-NN or BBox search)
    candidateWetlandIDs ← RTreeWetlands.QUERY_NEAREST(sightingPoint)
    /* Uses the R-tree to efficiently find wetlands whose MBRs are near the
       platypus sighting.                                                     */

    for each wetlandID in candidateWetlandIDs do
        currentWetlandPolygon ← LOOKUP_POLYGON(wetlandID)

        // b. Check for containment (Refine Stage 1: Point-in-Polygon)
        if POINT_IN_POLYGON(sightingPoint, currentWetlandPolygon) then
            minDistance ← 0
            foundInside ← true
            break
            // Sighting is inside, no need to check further for this sighting
        end
    end

    if not foundInside then
        // c. If not contained, calculate precise distance (Refine Stage 2:
           Point-to-Polygon)
        for each wetlandID in candidateWetlandIDs do
            currentWetlandPolygon ← LOOKUP_POLYGON(wetlandID)
            distance ← POINT_TO_POLYGON_DISTANCE(sightingPoint, currentWetlandPolygon)
            minDistance ← MIN(minDistance, distance)
        end
    end

    // d. Determine minimum distance and store results
    results.Add((sighting.ID, sighting.Lat, sighting.Lon, minDistance))
end
return results
```