1. Coordinate System for Pointy-Top Hexagonal Grids

To represent the cells in a pointy-top hexagonal grid, we can use one of the following systems:

- **Axial coordinates**: A two-dimensional system (q, r), where q represents the column, and r represents the row. This is intuitive for hex grids and makes distance calculations straightforward.
- **Cube coordinates**: A three-dimensional system (x, y, z) where x + y + z = 0. This system is great for distance calculation and identifying neighbors but requires slightly more storage.
- **Offset coordinates**: A modified two-dimensional system (row, col) often used for compatibility with rectangular grids.

Recommendation: Use **axial coordinates**, as they strike a balance between simplicity and computational efficiency.

2. Data Structures

- **Map Storage**:

- Use a **2D array** or a **hashmap** where the keys are the coordinates (q, r). A hashmap is preferable for sparse grids, while a 2D array is more efficient for dense grids.

- **Sensor Region Storage**:
    - For circles or rings, store the coordinates of all cells in the region in a **set**. Sets allow efficient intersection computation, which is central to this project.
    - Alternatively, if storage is a concern, use a **range representation** with mathematical boundaries and calculate intersections dynamically.

Recommendation: Use **sets** for simplicity and efficient intersection computation.

3. Best Data Structure to Store a Region Defined by Sensor Reading

A **region** defined by a sensor reading is a collection of hexagonal cells within a specific range from the sensor's center. The choice of data structure depends on how you plan to access and manipulate this data, especially for operations like intersections.

1. Suitable Data Structures

1. **HashSet<Hex>**:
    a. **What it is**: A set that stores unique `Hex` objects (coordinates) and allows efficient operations like insert, delete, and check for membership.
    b. **Why it's ideal**:
        i. Ensures uniqueness: No duplicate cells in the region.
        ii. Fast operations: Intersection and union can be performed efficiently using set operations.
        iii. Scalable: Handles sparse and irregularly shaped regions well.
    c. **Use case**: Ideal when the region needs to support intersection or union with other regions.
2. **2D Array/Grid**:
    a. **What it is**: A matrix-like structure where each cell corresponds to a specific hex cell on the map.
    b. **Why it's ideal**:
        i. Easy visualization: Directly represents the grid.
        ii. Efficient for dense regions: Best if the region spans a continuous block of cells.
    c. **Limitations**:
        i. Inefficient for sparse regions: Large memory overhead if most cells are empty.
        ii. Harder to compute intersections compared to sets.
    d. **Use case**: Useful if the entire map must be represented as a dense grid.

2. Circle vs. Ring Representation

- **Circle (within a distance)**:

    - All cells within a distance `d` from the center are part of the region.
    - HashSet<Hex> works perfectly because:
        - Calculating the cells in the range is straightforward.
        - The cells can be stored as unique elements in the set.
- **Ring (between two distances)**:
    - All cells within a specific range (e.g., `2 ≤ d ≤ 3`).
    - Storing such regions is also straightforward with `HashSet<Hex>`:
        - Generate all cells within the larger radius (`d = max`).
        - Remove cells within the smaller radius (`d = min`).
        - Store the result in a set.