1. Which distance metric is usable for distances between keys?

The **Manhattan Distance** is the most suitable metric for this problem.

- **Definition**: The Manhattan Distance between two keys on a grid is the sum of the absolute differences between their row and column positions.
- Formula: *d=|x1−x2|+|y1−y2|d = |x_1 - x_2| + |y_1 - y_2|*d=∣ x1 −x2 ∣ +∣ y1 −y2 ∣
- **Example**:
  - For a QWERTY keyboard:
    - Position of g: (row 3, column 4)
    - Position of d: (row 2, column 3)
    - Distance = *|3−2|+|4−3|=1+1=2|3 - 2| + |4 - 3| = 1 + 1 = 2*∣ 3−2∣ +∣ 4−3∣ =1+1=2

This metric is well-suited because:

1. It aligns with the way keys are laid out on a 2D grid.
2. It considers horizontal and vertical movements, which match the arrow-key-based navigation on virtual keyboards.

2. Would you need a particular data structure to represent the keyboard layout?

Yes, a **2D grid (array or list of lists)** is the best structure to represent the keyboard layout. Each key can be assigned a specific position based on its row and column.

Example Layout for a Virtual Keyboard:

```
1 2 3 4 5 6 7 8 9 0
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
```

Grid Representation:

```
Row 1: ['1', '2', '3', '4', '5', '6', '7', '8', '9', '0']
Row 2: ['q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p']
Row 3: ['a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l']
Row 4: ['z', 'x', 'c', 'v', 'b', 'n', 'm']
```

Permanent vs Temporary:

- **Permanent Structure**: Useful if distances need to be recalculated frequently or if the keyboard layout changes.
- **Temporary Structure**: Once distances are precomputed (e.g., mapping each key to valid moves), the 2D grid can be replaced with a dictionary or adjacency list for efficient lookups.

3. What Java data structure is best suited for mapping keys to valid moves?

The best Java data structure for mapping each key to a list of valid moves is a **HashMap** where:

- The key is the character (e.g., `'a'`).
- The value is a list of valid characters (e.g., `['s', 'z', 'q']`).

Why HashMap?

1. **Fast Lookup**: Key-to-value mapping is efficient with *O(1)O(1)*O(1) average time complexity.
2. **Scalability**: Can store mappings for all alphanumeric keys.
3. **Flexibility**: Easy to update if the keyboard layout changes.

4. Pseudocode for Creating an 8-Character Password

Pseudocode:

```
1. Define the keyboard layout as a 2D grid.
2. Create a function to compute Manhattan Distance between two keys.
3. Precompute valid moves for each key (keys with distance 2 or 3) and store in a HashMap.
4. Define a function `generatePassword(startKey)`:
    - Input: The starting key selected by the user.
    - Output: An 8-character password.
5. Initialize the password with the starting key.
6. Repeat until the password is 8 characters long:
    a. Get the current key.
    b. Retrieve the list of valid moves from the HashMap.
    c. Select a random key from the valid moves.
    d. Add the selected key to the password.
7. Return the final password.
```

Example Java-like Pseudocode:

```
// Function to generate an 8-character password
String generatePassword(char startKey, HashMap<Character,
List<Character>> validMoves) {
    StringBuilder password = new StringBuilder();
    password.append(startKey); // Start with the user-
selected key

    char currentKey = startKey;
    while (password.length() < 8) {
        // Get valid moves for the current key
        List<Character> moves =
validMoves.get(currentKey);

        // Select a random character from the valid moves
        char nextKey =
moves.get(randomIndex(moves.size()));

        // Add the selected key to the password
        password.append(nextKey);

        // Update the current key
        currentKey = nextKey;
    }
    return password.toString();
}
```

5. Compute the List of Valid Moves

Below is the list of valid moves for the keys `a`, `f`, `h`, `8`, `0`, and `p`:

Precomputed Valid Moves:

1. **Key `a`:**
    a. Valid Moves (2-3 distance): `['q', 'w', 's', 'z', 'x']`.
2. **Key `f`:**
    a. Valid Moves (2-3 distance): `['r', 't', 'g', 'v', 'c']`.
3. **Key `h`:**
    a. Valid Moves (2-3 distance): `['y', 'u', 'j', 'n', 'b']`.
4. **Key `8`:**
    a. Valid Moves (2-3 distance): `['5', '7', '9', 'i', 'k']`.
5. **Key `0`:**
    a. Valid Moves (2-3 distance): `['7', '8', '9', 'o', 'p']`.
6. **Key `p`:**
    a. Valid Moves (2-3 distance): `['o', 'l', 'm', '9', '0']`.

Final Summary

1. **Distance Metric:**
    a. Use Manhattan Distance for key-to-key distance calculations.
2. **Data Structure for Keyboard Layout:**
    a. Use a 2D array for temporary representation, and a HashMap for precomputed valid moves.
3. **Data Structure for Valid Moves:**

   a. HashMap is ideal for mapping keys to their valid moves.

4. **Password Generation Pseudocode:**
   a. Generate an 8-character password using valid moves from the HashMap.
5. **Precomputed Valid Moves:**
   a. The list of valid moves for keys `a`, `f`, `h`, `8`, `0`, and `p` is computed based on Manhattan Distance.