

```

1 fun findCageSize(catSizes: List<Int>): Int {
2     // จัดกลุ่มขนาดของแมวและตรวจสอบว่ามีแมวที่ไม่ได้อยู่ติดกัน
3     val unmatchedSizes = catSizes.withIndex().Iterable<IndexedValue<Int>>
4         .groupBy{ it.value, { it.index } } Map<Int, List<Int>>
5         .filter { (_, indices) ->
6             // ค้นหาคำขนาดที่มีแมวไม่ได้อยู่ติดกัน
7             indices.zipWithNext().any { (a, b) -> b != a + 1 }
8         }
9         .keys
10    // ส่งคืนขนาดที่ใหญ่ที่สุดในบรรดาแมวที่ไม่ได้อยู่ติดกัน
11    return unmatchedSizes.maxOrNull() ?: 0
12 }
13
14 fun main() {
15     // รับจำนวนแมวจากผู้ใช้ ถ้าไม่ใช่จำนวนเต็ม ให้แสดง "Invalid input" และจบการทำงาน
16     val n = readLine()?.toIntOrNull() ?: return println("Invalid input.")
17     // ตรวจสอบว่า n เป็นจำนวนเต็มหรือไม่
18     if (n % 2 != 0) { ... }
19     // อ่านขนาดของแมวทีละตัว
20     val catSizes = mutableListOf<Int>()
21     repeat(n) { i, Int
22         val size = readLine()?.toIntOrNull()
23         if (size != null) {
24             // ถ้า input เป็นจำนวนเต็ม ให้เพิ่มลงในรายการ catSizes
25             catSizes.add(size)
26         } else {
27             // ถ้า input ไม่ใช่จำนวนเต็ม ให้แสดง "Invalid input" และจบการทำงาน
28             return
29         }
30     }
31     // คำนวณขนาดคกรงที่เล็กที่สุดที่ต้องการสำหรับแมว
32     val cageSize = findCageSize(catSizes)
33     // แสดงขนาดคกรงที่เหมาะสม
34     println("$cageSize")
35 }

```

โค้ดนี้ใช้ **Functional Programming Paradigm** ในการแก้ไขปัญหา โดย Kotlin รองรับแนวคิดแบบฟังก์ชันและเป็นภาษาที่สามารถใช้เขียนเชิงฟังก์ชันได้ดี การเลือกใช้ฟังก์ชันเพื่อลดความซับซ้อนและทำให้โค้ดเข้าใจได้ง่ายขึ้นเป็นแนวคิดที่ดีในบริบทนี้

Paradigm ที่ใช้: Functional Programming

- **Declarative Style:** โค้ดในฟังก์ชัน `findCageSize` ใช้ฟังก์ชันสำเร็จรูปของ Kotlin เพื่อทำงานกับข้อมูล ทำให้เราไม่จำเป็นต้องระบุขั้นตอนละเอียด ลดการใช้คำสั่งแบบ procedural ซึ่งสอดคล้องกับการเขียนโปรแกรมเชิงฟังก์ชัน
- **Stateless Computations:** ฟังก์ชัน `findCageSize` ไม่ได้เปลี่ยนแปลงข้อมูลใน `catSizes` แต่จะรับข้อมูลมาและคืนค่าผลลัพธ์โดยไม่แก้ไขข้อมูลเดิม ซึ่งช่วยให้โค้ดอ่านง่ายขึ้นและลดปัญหาข้อผิดพลาดที่อาจเกิดขึ้นจากการเปลี่ยนแปลงข้อมูลภายนอก
- **Higher-Order Functions:** ฟังก์ชันที่ใช้ เช่น `groupBy`, `filter`, `zipWithNext`, และ `maxOrNull` เป็น higher-order functions ที่ใช้ในการจัดการข้อมูลในลักษณะของ functional programming ซึ่งช่วยให้โค้ดดูสะอาดและสั้นลง ไม่ต้องใช้การวนลูปเพื่อคำนวณแบบเดิม

แนวทางการแก้ปัญหาในโค้ด

โค้ดนี้มีขั้นตอนการแก้ปัญหาที่แบ่งเป็นลำดับ ดังนี้:

1. **รับ Input:** รับจำนวนแมว n จากผู้ใช้ ซึ่งจะต้องเป็นจำนวนเต็มคู่ หากไม่ใช่จำนวนเต็มคู่จะให้ผู้ใช้นำใหม่ จากนั้นรับขนาดของแมวทีละตัว (แต่ละบรรทัด) ไปเก็บใน List `catSizes`
2. **การประมวลผล (findCageSize):**
 - **Grouping:** ใช้ฟังก์ชัน `groupBy` เพื่อจัดกลุ่มแมวตามขนาด เพื่อที่จะระบุของแมวที่มีขนาดเท่ากัน
 - **Filtering Non-adjacent Pairs:** ใช้ `zipWithNext` เพื่อตรวจสอบว่ามีคู่แมวที่ไม่ได้อยู่ติดกันหรือไม่ หากพบว่าคู่แมวที่ไม่ติดกัน จะเก็บขนาดของแมวนั้นไว้ใน `unmatchedSizes`
 - **Find Maximum Size:** ใช้ `maxOrNull` เพื่อตรวจสอบขนาดที่ใหญ่ที่สุดในบรรดาแมวที่ไม่ได้อยู่ติดกัน หากไม่มีคู่ที่ต้องเคลื่อนย้าย ผลลัพธ์จะเป็น 0
3. **แสดงผลลัพธ์:** ขนาดของกรงที่เหมาะสมที่สุดจะถูกพิมพ์ออกมาในรูปแบบขนาดของกรงที่เล็กที่สุดที่สามารถเคลื่อนย้ายแมวได้ทั้งหมด โดยที่ช่วยให้แมวอยู่ติดกัน