# 1956. Minimum Time For K Virus Variants to Spread

## Problem Explanation

This problem presents us with a uniquely challenging task. There are n unique virus variants spread across an infinite 2D grid. Each virus variant is represented as a set of point coordinates (x, y) at day 0. Each day, every cell infected with a virus variant spreads the virus to all the neighbouring cells in four cardinal directions: up, down, left, and right. It's important to note that if a cell has multiple virus variants, all of them spread without interfering each other.

Our task is to find the minimum integer number of days it would take for any point to contain at least 'k' unique virus variants.

For instance, if we have points [[1,1],[2,2],[3,3]] and k = 3, the function should return 3. This is because on day 1, virus 1 will be at points (1,1),(2,2) and (3,3). On day 2, they will have spread reaching (3,3) and virus 2 and 3 also reaching (3,3) approximately on day 3.

## Solution Explanation

The solution uses an algorithm that iteration through possible points within a set upper limit (kMax), for each of these points, it measures and keeps the k smallest Manhattan distances from the virus points to this point (a,b). It then keeps track of the minimum of these k smallest distances.

The Manhattan distance between two points are calculated as `abs(x1 - x2) + abs(y1 - y2)` where (x1, y1) and (x2, y2) are the coordinates for the two points.

## Algorithm Steps

1. Set a value of kMax = 100, and a maximum value 'ans' (to record our minimum distance)
2. Now for each point (a, b) from (1, 1) to (kMax, kMax), do the following:
   a. For each virus variant point, calculate its distance from (a, b) and add it to a maximum heap.
   b. If the heap's size becomes more than k, we remove the maximum distance from it (it is of no use to us anymore)
   c. After checking all points, we compare the maximum value in the heap (which is the kth smallest distance) with our current 'ans'. Update 'ans' if this value is smaller.
3. Return 'ans' as this minima would give us the minimum days for a point to contain atleast k variants.

## Python Solution

```
python
import heapq

class Solution:

    def minDayskVariants(self, points, k):
        kMax = 100
        ans = float('inf')

        for a in range(1, kMax+1):
            for b in range(1, kMax+1):

                # Priority queue to store K smallest distances
                maxHeap = []

                for point in points:
                    # calculate and push the manhattan distances into heap
                    distance = abs(point[0] - a) + abs(point[1] - b)
                    heapq.heappush(maxHeap, -distance)

                    # if size > k, pop the maximum
                    if len(maxHeap) > k:
                        heapq.heappop(maxHeap)

                # update our minimum
                ans = min(ans, -maxHeap[0])
        return ans
```

Here, we make use of python's heapq module to implement heap operations. For Java, JavaScript, C#, and C++ solutions, we can follow similar approach while suitably using each languages' heap features. It is worth noting that use heap for this problem because we want to efficiently get the k smallest distances. We convert the distances into negative while pushing into the heap. This way, the maximum distance, which we want to pop off, always stays at the top of the heap. This is because heapq in python only supports min heap.# JavaScript Solution

In JavaScript, there's no built-in heap data structure. However, we can use an array to emulate it.

```
javascript
class Solution {
    constructor() {
        this.kMax = 100;
    }

    minDayskVariants(points, k) {
        let ans = Infinity;

        for (let a = 1; a < this.kMax+1; a++) {
            for (let b = 1; b < this.kMax+1; b++) {
                let maxHeap = [];

                for (let point of points) {
                    let distance = Math.abs(point[0] - a) + Math.abs(point[1] - b);
                    this.pushToHeap(maxHeap, -distance);
                    if (maxHeap.length > k) {
                        this.popFromHeap(maxHeap);
                    }
                }
                ans = Math.min(ans, -maxHeap[0]);
            }
        }
        return ans;
    }

    pushToHeap(heap, val) {
        heap.push(val);
        heap.sort((a, b) => b - a);
    }

    popFromHeap(heap) {
        heap.shift();
    }
}
```

The pushToHeap method sorts the heap in descending order every time an element is pushed. popFromHeap method removes the largest element (located in the 0 index due to sorting) from the heap.

## Java Solution

Java provides a PriorityQueue class that we can use as a max heap.

```
java
public class Solution {

    private static final int KMAX = 100;

    public int minDayskVariants(int[][] points, int k) {

        int ans = Integer.MAX_VALUE;
        for (int a = 1; a < KMAX+1; a++) {
            for (int b = 1; b < KMAX+1; b++) {

                // Max heap to store K smallest distances
                PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());

                for (int[] point : points) {
                    int distance = Math.abs(point[0] - a) + Math.abs(point[1] - b);
                    maxHeap.add(-distance);
                    if (maxHeap.size() > k) {
                        maxHeap.poll();
                    }
                }
                ans = Math.min(ans, -maxHeap.peek());
            }
        }
        return ans;
    }
}
```

Here, `Comparator.reverseOrder()` is used to create a max heap by reversing the natural order. Similar to the previous solutions, we push negative distances to the heap.