```swift
// Copyright © 2020 AlgoExpert, LLC. All rights reserved.

class Program {
  struct Item {
    var arrayIdx: Int
    var elementIdx: Int
    var num: Int
  }

  // O(nlog(k) + k) time | O(n + k) space - where where n is the total
  // number of array elements and k is the number of arrays
  static func mergeSortedArrays(_ arrays: [[Int]]) -> [Int] {
    var sortedList = [Int]()
    var smallestItems = [Item]()

    for arrayIdx in 0 ..< arrays.count {
      smallestItems.append(Item(arrayIdx: arrayIdx, elementIdx: 0, num: arrays[arrayIdx][0]))
    }

    var mh = MinHeap(array: smallestItems)
    while mh.length() != 0 {
      var smallestItem = mh.remove()
      sortedList.append(smallestItem.num)
      if smallestItem.elementIdx == arrays[smallestItem.arrayIdx].count - 1 {
        continue
      }
      mh.insert(Item(arrayIdx: smallestItem.arrayIdx, elementIdx: smallestItem.elementIdx + 1,
                     num: arrays[smallestItem.arrayIdx][smallestItem.elementIdx + 1]))
    }
    return sortedList
  }

  static func getMinValue(_ items: inout [Item]) -> Item {
    var minValueItem = items[0]
    for i in 1 ..< items.count {
      if items[i].num < minValueItem.num {
        minValueItem = items[i]
      }
    }
    return minValueItem
  }

  class MinHeap {
    var heap = [Item]()

    init(array: [Item]) {
      heap = array
      buildHeap(array: array)
    }

    // O(n) time | O(1) space
    func buildHeap(array: [Item]) {
      var firstParentIndex = Double((array.count - 2) / 2)
      firstParentIndex = firstParentIndex.rounded(.down)

      for var currentIndex in (0 ... Int(firstParentIndex)).reversed() {
        var endIndex = array.count - 1
        siftDown(currentIndex: currentIndex, endIndex: endIndex)
      }
    }

    // O(log(n)) time | O(1) space
    func siftDown(currentIndex: Int, endIndex: Int) {
      var childOneIdx = currentIndex * 2 + 1
      var current = currentIndex
      while childOneIdx <= endIndex {
        var childTwoIdx = -1
        if current * 2 + 2 <= endIndex {
          childTwoIdx = current * 2 + 2
        }
        var indexToSwap = childOneIdx
        if childTwoIdx > -1, heap[childTwoIdx].num < heap[childOneIdx].num {
          indexToSwap = childTwoIdx
        }

        if heap[indexToSwap].num < heap[current].num {
          swap(firstIndex: current, secondIndex: indexToSwap)
          current = indexToSwap
          childOneIdx = current * 2 + 1
        } else {
          return
        }
      }
```

```
    }

    // O(log(n)) time | O(1) space
    func siftUp() {
      var currentIndex = heap.count - 1
      var parentIndex = (currentIndex - 1) / 2

      while currentIndex > 0 {
        var current = heap[currentIndex].num
        var parent = heap[Int(parentIndex)].num
        if current < parent {
          swap(firstIndex: currentIndex, secondIndex: parentIndex)
          currentIndex = parentIndex
          parentIndex = (currentIndex - 1) / 2
        } else {
          return
        }
      }
    }

    // O(1) time | O(1) space
    func peek() -> Item {
      return heap[0]
    }

    // O(log(n)) time | O(1) space
    func remove() -> Item {
      var l = heap.count
      swap(firstIndex: 0, secondIndex: l - 1)
      var peeked = heap[l - 1]
      heap.removeLast()
      siftDown(currentIndex: 0, endIndex: l - 2)
      return peeked
    }

    // O(log(n)) time | O(1) space
    func insert(_ value: Item) {
      heap.append(value)
      siftUp()
    }

    // Generic swap function
    func swap(firstIndex: Int, secondIndex: Int) {
      let temp = heap[firstIndex]
      heap[firstIndex] = heap[secondIndex]
      heap[secondIndex] = temp
    }

    func length() -> Int {
      return heap.count
    }
  }
}

```