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

package main

type Item struct {
  ArrayIdx   int
  ElementIdx int
  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
func MergeSortedArrays(arrays [][]int) []int {
  sortedList := []int{}
  smallestItems := []Item{}

  for arrayIdx := 0; arrayIdx < len(arrays); arrayIdx++ {
    smallestItems = append(smallestItems, Item{
      ArrayIdx:   arrayIdx,
      ElementIdx: 0,
      Num:        arrays[arrayIdx][0],
    })
  }

  mh := NewMinHeap(smallestItems)
  for mh.length() != 0 {
    smallestItem := mh.Remove()
    sortedList = append(sortedList, smallestItem.Num)
    if smallestItem.ElementIdx == len(arrays[smallestItem.ArrayIdx])-1 {
      continue
    }
    mh.Insert(Item{
      ArrayIdx:   smallestItem.ArrayIdx,
      ElementIdx: smallestItem.ElementIdx + 1,
      Num:        arrays[smallestItem.ArrayIdx][smallestItem.ElementIdx+1],
    })
  }
  return sortedList
}

type MinHeap []Item

func NewMinHeap(array []Item) *MinHeap {
  heap := MinHeap(array)
  ptr := &heap
  ptr.BuildHeap(array)
  return ptr
}

// O(n) time | O(1) space
func (h *MinHeap) BuildHeap(array []Item) {
  first := (len(array) - 2) / 2
  for currentIndex := first + 1; currentIndex >= 0; currentIndex-- {
    h.siftDown(currentIndex, len(array)-1)
  }
}

// O(log(n)) time | O(1) space
func (h *MinHeap) siftDown(currentIndex, endIndex int) {
  childOneIdx := currentIndex*2 + 1
  for childOneIdx <= endIndex {
    childTwoIdx := -1
    if currentIndex*2+2 <= endIndex {
      childTwoIdx = currentIndex*2 + 2
    }
    indexToSwap := childOneIdx
    if childTwoIdx > -1 && (*h)[childTwoIdx].Num < (*h)[childOneIdx].Num {
      indexToSwap = childTwoIdx
    }
    if (*h)[indexToSwap].Num < (*h)[currentIndex].Num {
      h.swap(currentIndex, indexToSwap)
      currentIndex = indexToSwap
      childOneIdx = currentIndex*2 + 1
    } else {
      return
    }
  }
}

// O(log(n)) time | O(1) space
func (h *MinHeap) siftUp() {
  currentIndex := h.length() - 1
  parentIndex := (currentIndex - 1) / 2
```

```
  for currentIndex > 0 {
    current, parent := (*h)[currentIndex].Num, (*h)[parentIndex].Num
    if current < parent {
      h.swap(currentIndex, parentIndex)
      currentIndex = parentIndex
      parentIndex = (currentIndex - 1) / 2
    } else {
      return
    }
  }
}

// O(log(n)) time | O(1) space
func (h *MinHeap) Remove() Item {
  l := h.length()
  h.swap(0, l-1)
  peeked := (*h)[l-1]
  *h = (*h)[0 : l-1]
  h.siftDown(0, l-2)
  return peeked
}

// O(log(n)) time | O(1) space
func (h *MinHeap) Insert(value Item) {
  *h = append(*h, value)
  h.siftUp()
}

func (h MinHeap) swap(i, j int) {
  h[i], h[j] = h[j], h[i]
}

func (h MinHeap) length() int {
  return len(h)
}
```