CS 3310
Homework 3
Mariia Kravtsova

Problem: Create an efficient iterative solution of merge sort.

Solution: For this solution I have a bottom up merge sort. Create small subarrays then merge them in pairs. Theoretically the bottom up merge sort uses between 1/2nlogn and nlogn accesses to sort an array of length n. This sort seems to work well for linked list since we start with smaller chunks of data and it is easier to compare linked list on a smaller scale. This solution is based on the solution provided in Algorithms 4th edition by Segwick.

```
MergeSort() {
  for (size = 1; size < length; size += size)
    for (index = 0; index < length-size; index += size+size)
      merge(array, index, index+size-1, minimum(index+size+size-1, length-1));
}

merge(array, low, mid, high) {
     i = low
     j = mid+1
    while (i < low.length ll j < high.length) {
        if (i < low.length && j < high.length) {
            if (low[i] <= high[j]) {
                array[k++] = low[i++];
            } else {
                array[k++] = high[j++];
            }
        } else if (i < low.length) {
            array[k++] = low[i++];
        } else if (j < high.length) {
            array[k++] = high[j++];
        }
    }
}
```

Problem: Write an algorithm of an iterative or recursive merge sort that will sort an array A[0… n-1] with n elements using n threads.

Solution: After trying the iterative approach I was able to chunk the array and start a thread for each. However after encountering a problem with merge I switched to the idea of using recursive merge sort but with my previous solution. Now instead of chunking data, I am chunking the threads. So I create to threads left and right, and from each of those I create another thread from left and right, and run them in parallel withe the data divided into two. This lets my merge run smoother than in the iterative approach. Since I did not know how threads or parallel sorting works I had to refer to the youtube video provided by Udacity, https://www.youtube.com/watch?v=_XOZ2IiP2nw

```
Main() {
  Array = [0 ... n]
  int size = length(Array)
  numberOfThreads = x * 2 // x is the desired number of cores, grows by 2

  for (i = 1; i <= numberOfThreads; i++) {
    ParallelMergeSort(Array, x, size)
  }
}

ParallelMergeSort(Array, x, size) {
  middle = size / 2
  if x == 0 {
    mergeSort(Array)
  }
  // chunk data
  leftArray = Arrays.copy(0, middle) // copy from 0 to middle of the original Array
  rightArray = Arrays.copy(middle, size) // copy from middle to end of the original Array

  // Create leftThread and right thread that run MergeSort
  leftThread = Thread(RunnableMergeSort(leftArray, x / 2))
  rightThread = Thread(RunnableMergeSort(rightArray, x / 2))

  // Start the threads
  leftThread.start()
  rightThread.start()

  // Wait for all of them to finish
  leftThread.join()
  rightThread.join()

  // Merge the arrays
  merge(left, right, Array, size)
}

// Runnable runs the ParallelMergeSort which repeats the division
// This creates layers of parallel threads
RunnableMergeSort (Array, threadLevel) {
  ParallelMergeSort(Array, threadLevel)
}

Merge(left, right, Array, size) {
  for (i = 0; i < size; i++) {
    if b >= length(right)
      Array[i] = left[a]
      a++
    if a < length(left) and left[a] < right[b]
      Array[i] = left[a]
      a++
```

```
    else
      Array[i] = right[b]
      b++
  }
}

// Below is a less working solution for iterative approach

Main() {
  Array = [0 ... n]
  tempArray
  size = length(Array)

  chunk_size = array.length / threadNumber;
  for (start = 0; start < array.length ; start+=chunk_size) {
    tempArray = Arrays.copy(start, start + chunk_size);
    Thread NameThread = new Thread(new MergeSort(tempArray)
  }
}

MergeSort(tempArray) {
  for (size = 1; size < length; size += size)
    for (index = 0; index < length-size; index += size+size)
      merge(tempArray, index, index+size-1, minimum(index+size+size-1, length-1));
}

merge(array, low, mid, high) {
     i = low
     j = mid+1
     // Copy low to high
     for (k = low; k <= high; k++)
        temp[k] = array[k]

     // Merge array low to high
     for (k = low; k <= high; k++)
        if      (i > mid)
          array[k] = temp[j++]
        else if (j > high )
          array[k] = temp[i++]
        else if (temp[j] < temp[i])
          array[k] = temp[j++]
        else
          array[k] = temp[i++]
}
```

I do NOT give permission to the instructor to share my solution(s) with the class.