Merge Sort

Algorithm:

Merge Step:

Merge two sorted lists into one sorted list in linear time.

Let Q1 and Q2 be two sorted queues.
Let Q be an empty queue.

// One of these queues will eventually empty out and
// break from loop
while (Q1 and Q2 are not empty) {
item1 = Q1.front();
item2 = Q2.front();

Add min(item1, item2) from (Q1 or Q2) to end of Q
}

concatenate non-empty(Q1, Q2) to end of Q

Actual Algorithm:

Start with unsorted list I of n items
Break into two lists with middle index (**I1**, **I2**)
Sort I1 recursively, yielding S1
Sort I2 recursively, yielding S2
Merge S1 and S2 into one sorted list S

Properties:

- O(n log(n)) time
- O(n) memory required for arrays (merge step)
  - O(log(n)) memory for recursion stack space but merge step memory dominates the stack space
- O(log(n)) memory for linked list (merge can be done in O(1))
- Recursive divide-and-conquer algorithm
  - Dividing simple
  - Do all working merging step
- Natural for Linked Lists (In-Place)
  - QuickSort requires pivot, this does not.
  - MergeSort doesn't require random access like QuickSort does.
- Not In-Place Sort For Arrays
- Stable Sorting Algorithm
- Great if numbers larger than memory
  - External Merge Sort
- Worse Locality of Reference Than QuickSort

```
public static void mergeSort(int arr[]) {
  mergeSort(arr, new int[arr.length], 0, arr.length - 1);
}
/** The fullCopy variable is the O(n) memory **/
private static void mergeSort(int arr[], int fullCopy[], int low, int high) {
  if (low < high) {
    int mid = (low + high) / 2;
    // Sort Left Half Infinitely [low, mid]
    mergeSort(arr, fullCopy, low, mid);
    // Sort Right Half Infinitely [mid + 1, high]
    mergeSort(arr, fullCopy, mid + 1, high);
    // Merge Two Sorted Halves
    merge(arr, fullCopy, low, mid, high);
  }
}
```

```
private static void merge(int arr[], int fullCopy[], int low, int mid, int high) {
  // Copy Array From [low, high] into fullCopy[]
  System.arraycopy(arr, low, fullCopy, low, high - low + 1);

  int leftPtr = low;
  int rightPtr = mid + 1;

  int current = low;

  // Grabbing smallest from left and right halves of the
  // array and putting them into the original array
  while(leftPtr <= mid && rightPtr <= high){
    if(fullCopy[leftPtr] <= fullCopy[rightPtr]){
      arr[current] = fullCopy[leftPtr];
      leftPtr++;
    } else {
      arr[current] = fullCopy[rightPtr];
      rightPtr++;
    }
    current++;
  }

  // Finishing off remaining left over array parts
  while(leftPtr <= mid){
    arr[current++] = fullCopy[leftPtr++];
  }
  while(rightPtr <= high){
    arr[current++] = fullCopy[rightPtr++];
  }
}
```

Merge Sort - Spreadsheet

Merge Algorithm

							Current	Left Ptr	Right Ptr	
Merge Algorithm		3	7	2	8					
		1. Copy Array Into FullCopy								
Original Array		3	7	2	8		0	0	2	Since 2 < 3, set arr[current] = 2 and increment (rightPtr, current)
Full Copy Array		3	7	2	8					
		Left Pointer		Right Pointer						
Original Array		2	7	2	8		1	1	2	Since 3 < 8, set arr[current] = 3 and increment (leftPtr, current)
Full Copy Array		3	7	2	8					
		Left Pointer			Right Pointer					
Original Array		2	3	2	8		2	1	3	Since 7 < 8, set arr[current] = 7 and increment (leftPtr, current)
Full Copy Array		3	7	2	8					
			Left Pointer		Right Pointer					
Original Array		2	3	7	8		3	2	3	
				Left Pointer	Right Pointer					
		Wait! Left pointer is out of bounds. We now just copy what is left from rightPtr to end of array. Coincendence, it is already sorted.								