Merge sort

- Use a divide-and-conquer approach:
  1. If list is of length 0 or 1, already sorted
  2. If list has more than one element, split into two lists, and sort each
  3. Merge results
     1. To merge, just look at first element of each, move smaller to end of the result
     2. When one list empty, just copy rest of other list

Example of merging

Left in list 1	Left in list 2	Compare	Result
[1,5,12,18,19,20]	[2,3,4,17]	1, 2	[]
[5,12,18,19,20]	[2,3,4,17]	5, 2	[1]
[5,12,18,19,20]	[3,4,17]	5, 3	[1,2]
[5,12,18,19,20]	[4,17]	5, 4	[1,2,3]
[5,12,18,19,20]	[17]	5, 17	[1,2,3,4]
[12,18,19,20]	[17]	12, 17	[1,2,3,4,5]
[18,19,20]	[17]	18, 17	[1,2,3,4,5,12]
[18,19,20]	[]	18, --	[1,2,3,4,5,12,17]
[]	[]		[1,2,3,4,5,12,17,18,19,20]

Complexity of merge

- Comparison and copying are constant
- Number of comparisons – O(len(L))
- Number of copyings – O(len(L1) + len(L2))
- So merging is linear in length of the lists

```
def merge(left, right, compare):
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if compare(left[i], right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

Putting it together

```
import operator

def mergeSort(L, compare = operator.lt):
    if len(L) < 2:
        return L[:]
    else:
        middle = int(len(L)/2)
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        return merge(left, right, compare)
```

Complexity of merge sort

- Merge is O(len(L))
- Mergesort is O(len(L)) * number of calls to merge
  - O(len(L)) * number of calls to mergesort
  - O(len(L) * log(len(L)))
- Log linear – O(n log n), where n is len(L)
- Does come with cost in space, as makes new copy of list