Merge Sort
Slow Sorting versus Fast Sorting

- Algorithms such as *selection sort*, *insertion sort*, *bubble sort*, etc. are all extremely slow for large lists.

- This is because they take $N^2$ time on a list of size $N$.

- Algorithms that are based on “divide-and-conquer” such as *merge sort* or *quick sort* are much faster.

- These algorithms run in $N \log N$ on a list of size $N$. 
Divide-and-Conquer

- **Divide step**: Partition the problem into sub-problems.

- **Conquer step**: Solve each sub-problem separately.

- **Combine step**: Combine the solutions of the sub-problems into a solution of the original problem.
Merge Sort: Main Idea

- **Divide Step:** Partition the list into two halves.

- **Conquer Step:** Sort each of the halves separately.

- **Combine Step:** “Merge” the two sorted halves into a sorted whole.
# The merge sort function; sorts the sublist L[first:last+1]
def generalMergeSort(L, first, last):
    # Base case: if first == last then it is already sorted

    # Recursive case: L[first:last+1] has size 2 or more
    if first < last:
        # divide step
        mid = (first + last)/2

        # conquer step
        generalMergeSort(L, first, mid)
        generalMergeSort(L, mid+1, last)

        # combine step
        merge(L, first, mid, last)
Sometimes for recursive functions we have to create extra parameters that control the recursion.

We saw this in binarySearch and we see this again in mergeSort.

However, the calling code should not be expected to add these extra parameters.

So such recursive functions usually come with “wrapper functions.”

```python
# Wrapper function
def mergeSort(L):
    generalMergeSort(L, 0, len(L)-1)
```
The Merge Function: Main Idea

- **Problem**: we are given two lists, both sorted, and we want to merge these into a single sorted list.

- Let us call these lists $L_1$ and $L_2$. Let $p_1$ be an index pointing to the first slot of $L_1$ and $p_2$ be an index pointing to the first slot of $L_2$.

- Compare $L[p_1]$ and $L_2[p_2]$. Whichever element is smaller is the “winner” and is “output.”

- The index in the winning list is then incremented.
The Merge Function: Main Idea

3 is the winner

6 is the winner

7 is the winner
The Merge Function

# Assumes that L[first:mid+1] is sorted and also
# that L[mid: last+1] is sorted. Returns L with L[first: last+1] sorted

def merge(L, first, mid, last):
    i = first # index into the first half
    j = mid + 1 # index into the second half

    tempList = []

    # This loops goes on as long as BOTH i and j stay within their
    # respective sorted blocks
    while (i <= mid) and (j <= last):
        if L[i] <= L[j]:
            tempList.append(L[i])
            i += 1
        else:
            tempList.append(L[j])
            j += 1
The Merge Function: Continued

# If i goes beyond the first block, there may be some elements
# in the second block that need to be copied into tempList.
# Similarly, if j goes beyond the second block, there may be some
# elements in the first block that need to be copied into tempList

if i == mid + 1:
    tempList.extend(L[j:last+1])
elif j == last + 1:
    tempList.extend(L[i:mid+1])

L[first:last+1] = tempList
# print tempList