Merge Sort

APRIL 24TH, 2013
Merge Sort Algorithm

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

- **Conquer Step**
  Separately sort each half.

- **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)
- No comparisons are being made in this function.
- No movement of elements happens in this function.
- All such work happens in the `merge` function we are yet to develop.
- However, the `generalMergeSort` function is extremely important because it organizes and parcels out work to the `merge` function.
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 $L_2$. Initialize index $p_1$ to point to the first slot of $L_1$ and index $p_2$ to point 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.
Illustration

3 7 11 12 17

6 14 14 16 22 70

3 is the winner

3 7 11 12 17

6 14 14 16 22 70

6 is the winner

3 7 11 12 17

6 14 14 16 22 70

7 is the winner
More Details

- The `merge` function is called as:
  \[
  \text{merge}(L, \text{first}, \text{mid}, \text{last})
  \]

- The two halves are \(L[\text{first} : \text{mid}]\) and \(L[\text{mid}+1 : \text{last}+1]\).

- We need an index to start off at `first` and an index to start off at `mid+1`. 
• One of the indices may quickly reach the end of its half.

3 4 6 10 13 16

12 18 64 71 80 96

• In this example, the competitions are: 3 vs 12, 4 vs 12, 6 vs 12, 10 vs 12, 13 vs 12, 13 vs 18, and finally 16 vs 18.

• 16 is the “winner” and the first index moves past the end of the first half.
# 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 = []  # output list

    # 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
# 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])

# Finally move everything back from tempList into the appropriate
# slice of L
L[first:last+1] = tempList
Playing with merge sort

- By inserting print statements at strategic locations, you can start understanding how merge sort works.

- For example, if we insert `print L[first:last+1]` right at the beginning of the function, here is what we see:

```
L = [3, 1, 2, 4, 1, -1]
genralMergeSort(L, 0, 5)
[3, 1, 2, 4, 1, -1]
[3, 1, 2]
[3, 1]
[3]
[1]
[2]
[4, 1, -1]
[4, 1]
[4]
[1]
[-1]
```
By inserting `print L[first:last+1]` right at the end of the function, we get a different view of what is going on.

```
L = [3, 1, 2, 4, 1, -1]
generalMergeSort(L, 0, 5)
[1, 3]
[1, 2, 3]
[1, 4]
[-1, 1, 4]
[-1, 1, 1, 2, 3, 4]
```
Why is merge sort efficient?

- The slow sorting algorithms essentially compare all pairs of elements.
- Say we have a list of size $N$. So each half has size $N/2$.
- Hence, there are $N^2/4$ pairs of elements with one element in the first half and one in the second.
- In merge sort, elements from the first half are not compared with elements from the second half, *until both halves are sorted.*
- This leads to a huge reductions in the number of comparisons we need to make.
Why is merge sort so efficient?

- In the merge function, if \( L[i] < L[j] \), then \( L[i] \) and does not have to compared to \( L[j+1], L[j+2] \), etc.

- More specifically, every time we make a comparison in merge, an index moves forward one step.

- The two indices can, together, travel a distance of at most \( N \).

- So the merge function makes \( N \) comparisons.

**Example:** \( N = 2000 \). The merge function makes 2000 comparisons, whereas a slow sort makes \((2000) \times (2000)/4 = 1000,000\) comparisons.
Comparing merge sort and selection sort

- We have posted a program called `mergeSortTiming.py` that times merge sort and selection sort and compares the running times of these.
- The program constructs a list of 100,000 randomly chosen integers and sends this list to merge sort and a copy of this list to selection sort.
- A recent run of this program yielded:
  
  0.780428886414 seconds for merge sort
  304.80786109 seconds for merge sort