The Merge Sort

We now turn our attention to using a divide and conquer strategy as a way to improve the performance of sorting algorithms. The first algorithm we will study is the **merge sort**. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Figure 10 shows our familiar example list as it is being split by `mergeSort` . Figure 11 shows the simple lists, now sorted, as they are merged back together.

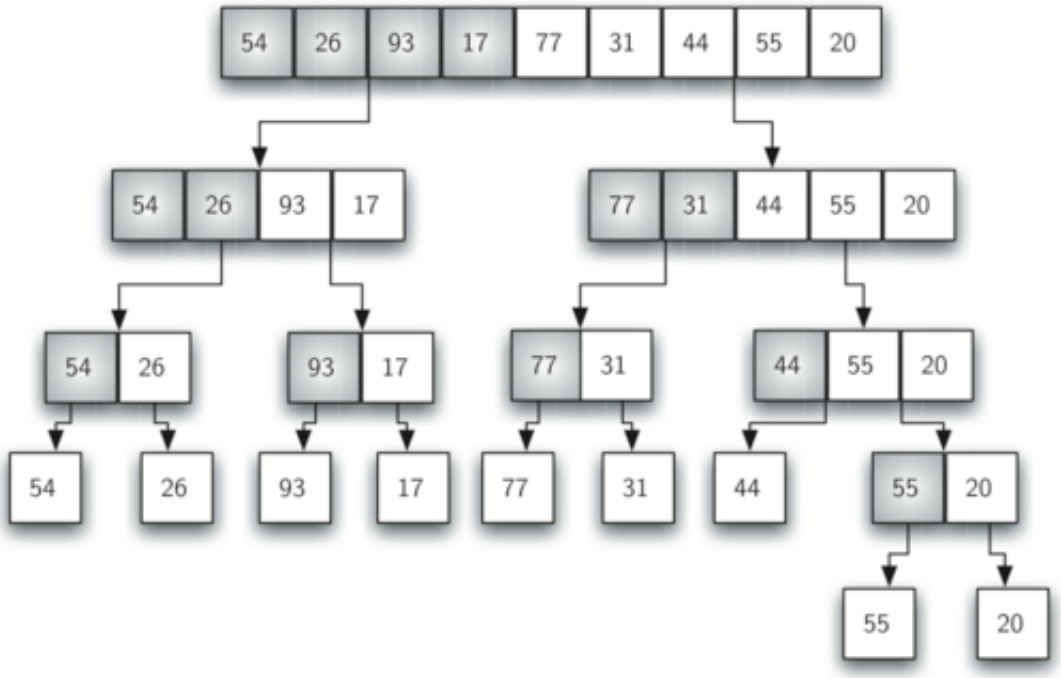


Figure 10: Splitting the List in a Merge Sort

ellSort.html)

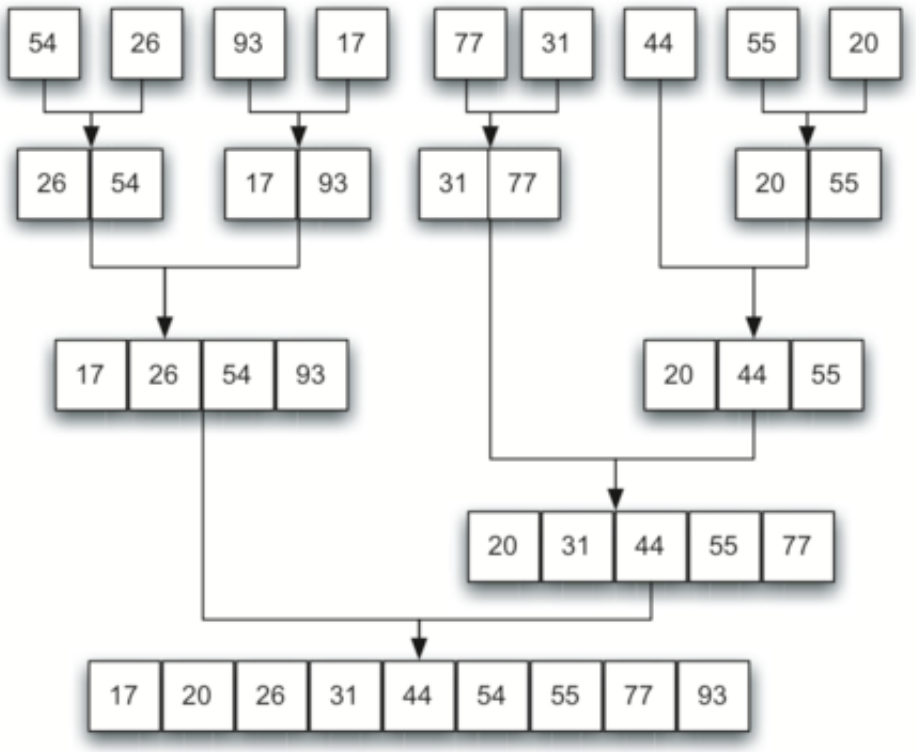


Figure 11: Lists as They Are Merged Together

The `mergeSort` function shown in ActiveCode 1 begins by asking the base case question. If the length of the list is less than or equal to one, then we already have a sorted list and no more processing is necessary. If, on the other hand, the length is greater than one, then we use the Python `slice` operation to extract the left and right halves. It is important to note that the list may not have an even number of items. That does not matter, as the lengths will differ by at most one.

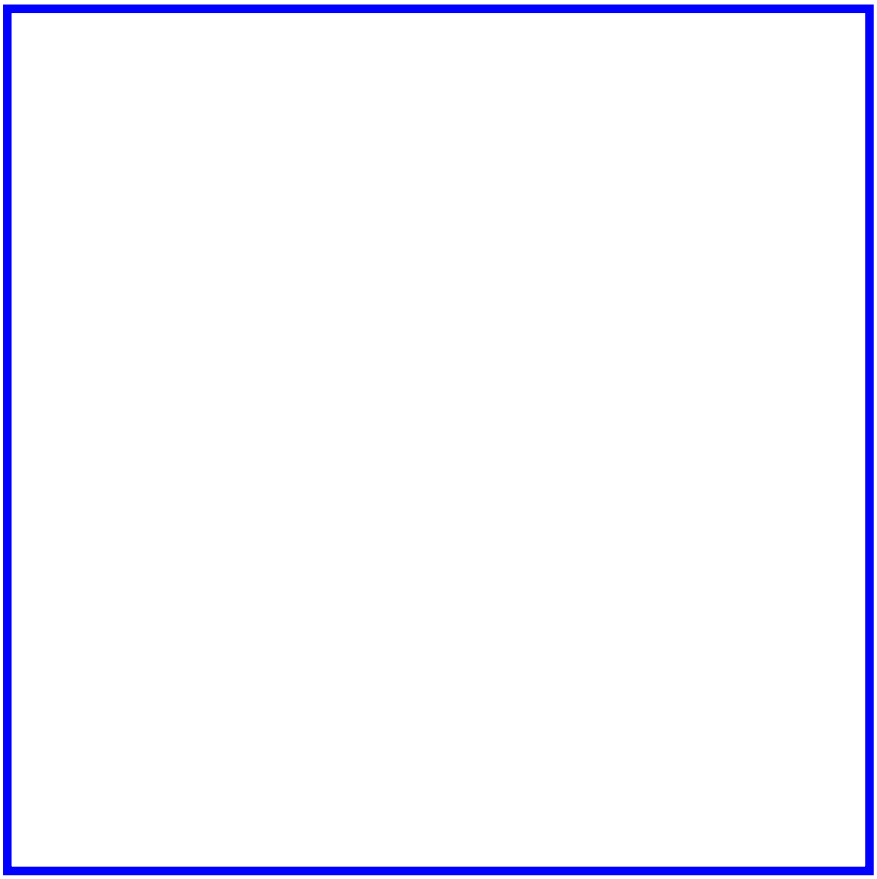
Run Save Load Show CodeLens

```
 1 def mergeSort(alist):
 2     print("Splitting ",alist)
 3     if len(alist)>1:
 4         mid = len(alist)//2
 5         lefthalf = alist[:mid]
 6         righthalf = alist[mid:]
 7 
 8         mergeSort(lefthalf)
 9         mergeSort(righthalf)
10 
11         i=0
12         j=0
13         k=0
14         while i < len(lefthalf) and j < len(righthalf):
15             if lefthalf[i] < righthalf[j]:
16                 alist[k]=lefthalf[i]
17                 i=i+1
18             else:
19                 alist[k]=righthalf[j]
20                 j=j+1
21             k=k+1
22 
23         while i < len(lefthalf):
24             alist[k]=lefthalf[i]
25             i=i+1
```

ActiveCode: 1 Merge Sort (lst_mergeSort)

Once the `mergeSort` function is invoked on the left half and the right half (lines 8–9), it is assumed they are sorted. The rest of the function (lines 11–31) is responsible for merging the two smaller sorted lists into a larger sorted list. Notice that the merge operation places the items back into the original list (`alist`) one at a time by repeatedly taking the smallest item from the sorted lists.

The `mergeSort` function has been augmented with a `print` statement (line 2) to show the contents of the list being sorted at the start of each invocation. There is also a `print` statement (line 32) to show the merging process. The transcript shows the result of executing the function on our example list. Note that the list with 44, 55, and 20 will not divide evenly. The first split gives [44] and the second gives [55,20]. It is easy to see how the splitting process eventually yields a list that can be immediately merged with other sorted lists.



Initialize Run Stop
Beginning Step Forward Step Backward End

In order to analyze the `mergeSort` function, we need to consider the two distinct processes that make up its implementation. First, the list is split into halves. We already computed (in a binary search) that we can divide a list in half $\log n$ times where *n* is the length of the list. The second process is the merge. Each item in the list will eventually be processed and placed on the sorted list. So the merge operation which results in a list of size *n* requires *n* operations. The result of this analysis is that $\log n$ splits, each of which costs n for a total of $n \log n$ operations. A merge sort is an $O(n \log n)$ algorithm.

Recall that the slicing operator is $O(k)$ where k is the size of the slice. In order to guarantee that `mergeSort` will be $O(n \log n)$ we will need to remove the slice operator. Again, this is possible if we simply pass the starting and ending indices along with the list when we make the recursive call. We leave this as an exercise.

It is important to notice that the `mergeSort` function requires extra space to hold the two halves as they are extracted with the slicing operations. This additional space can be a critical factor if the list is large and can make this sort problematic when working on large data sets.

Self Check

Q-44: Given the following list of numbers:
 [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43,

34, 46, 40]
 which answer illustrates the list to be sorted after 3 recursive calls to mergesort?

- [16, 49, 39, 27, 43, 34, 46, 40]
- [21,1]
- [21, 1, 26, 45]
- [21]

Check Me Compare me

Q-45: Given the following list of numbers:
 [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]
 which answer illustrates the first two lists to be merged?

- [21, 1] and [26, 45]
- [[1, 2, 9, 21, 26, 28, 29, 45] and [16, 27, 34, 39, 40, 43, 46, 49]
- [21] and [1]
- [9] and [16]

Check Me Compare me

❮ (TheShellSort.html) ❯ (TheQuickSort.html)

user not logged in