Merge-Sort Lab - Reflection

Ryan Blocker
CS320 - Craig Partridge
February 20th, 2024

Code: He is the algorithm I submitted to Zybooks

```
def mergesort(mlist):

      if mlist is None:
      return None
      if not isinstance(mlist, list):
      return None
      if len(mlist) == 0:
      return []

      if (len(mlist) > 1):
            mid = len(mlist) // 2
            right_side = mlist[mid:]
            left_side = mlist[:mid]

            mergesort(right_side)
            mergesort(left_side)

            right_index = left_index = mlist_index = 0

            while (left_index < len(left_side) and right_index <
      len(right_side)):
                  if right_side[right_index] <= left_side[left_index]:
                        mlist[mlist_index] = right_side[right_index]
                              right_index += 1
                  else:
                        mlist[mlist_index] = left_side[left_index]
                        left_index += 1
                  mlist_index += 1

            while right_index < len(right_side):
                  mlist[mlist_index] = right_side[right_index]
                  right_index += 1
                  mlist_index += 1

            while left_index < len(left_side):
                  mlist[mlist_index] = left_side[left_index]
                  left_index += 1
                  mlist_index += 1
      return mlist
```

Approach: As you can see from my code above the chosen approach was recursion. I had tried several other iterative methods. For example, my first attempt I converted `mlist` into a list of lists and then I wanted to basically merge the other lists in `mlist` into one of the nested lists iteratively by checking whether the element I was currently looking at was less than the front or tail of the destination list and then if they were equal I had it search through the destination list to find the correct index for insertion. This worked fine until it for some reason didn't work for odd-numbered lists as well as the whole routine was $O(n^2)$ and not O(n log n) so I had to scrap the idea.

I normally shy away from recursion because it makes my head hurt trying to visualize all of the different iterations that function goes through so this final algorithm took me a while to fine tune to get the correct functionality. My solution does follow the `merge-sort` "methodology" if you will. I first divide the given list in half and assign both sides to a sub-list (`left_side` and `right_side`). Then I recursively call the `merge_sort` function with `left_side` and `right_side` to split up the given list into single element lists which by nature are already sorted. Then after the list has been spliced up it goes into the first while loop where it checks if the elements in the right_side are less than or equal to the left_side at the current index. If it is then set the element at the current `mlist` index to the current right_side index then increment the index. If that is not true then just add it to the left_side. After that is finished and to catch any remaining elements in the left and right side is what the last two while loops are for.

Modularity: As far as modularity in this algorithm goes it was a bit difficult to splice up the problem. Initially I had a convert_list function that took `mlist` and made it into a list of lists but since I had to scrap that idea it never made it to the final algorithm. I pasted it below in case you wanted to see it:

```
def convert_list(list):
     return [[el] for el in list]
```

In the final algorithm, I split up by a few things. First I get the edge cases out of the way, then I divide up the list, then finally I sort it and then add any remaining elements to the list.

Big-Oh Complexity: This was very tricky to get to be O(n log n) instead of $O(n^2)$. But if we break my algorithm down it is indeed now O(n log n).

We achieve this by dividing the list up into two halves at each single recursive step, which takes log n steps, and then merging the sublists takes O(n) (linear time). Therefore, the total time complexity is n times log n which is O(n log n).