Austin Youngren

CS-320-001

Lab 3: Merge Sort

02/20/2024

Algorithm

For my merge sort algorithm, I utilized recursion to split the input list, then utilized an iterative approach for comparing values. To implement this in a clean and easy to read manner, I created three subroutines: separate_list(mlist), sort_split(mlist), and sort_comparison(left_sect, right_sect, mlist). These subroutines helped me by reducing the visual complexity this algorithm would otherwise have if placed in one function. This helped me to get rid of duplicated code where I split the list, along with locating bugs within the code quicker than I would have otherwise. More specifically, it helped me find my issue with accidental duplications and deletions which is mentioned in further detail in later section.

The core function, mergesort(mlist), is simple and is mainly used to check for the cases where the input list is empty or None. If empty, the function will return an empty list; if None, the function will return None. If the input list passes these checks, then it will be passed to the subroutine sort_split() as a parameter, where the list's length will be checked. Since we already checked for an empty list prior to the call, the conditional will only be skipped when list_length = 1, which results in the return of mlist. When list_length > 1, mlist is divided in half (left_sect & right_sect), using separate_list(), then each half is sent as a parameter to sort_split(), resulting in a recursive call. This process continues until recursively divided sub-lists lengths are 1 and return mlist.

After returning, for each recursive instance, the left and right sections of mlist, along with mlist, are sent as parameters to the sort_comparison() subroutine. This subroutine creates 3 index values, i, j, and n and assigns them a value of zero; then, creates 2 variables, left_length and right_length, that are assigned the length value of left_sect and right_sect. After this, we enter the first while loop, which stops looping when i = left_length or j = right_length. Within the loop, left_sect[i] is compared to right_sect[j]. If left_sect[i] is less than or equal to right_sect[j], then the value at left_sect[i] is assigned to the element at index n of mlist, then i is incremented by 1; otherwise, the value of right_sect[j] is assigned to the element at index n of mlist, then j is incremented by 1. n is incremented by 1, after the if-else block, before the next iteration of the loop, to ensure future values don't overwrite the currently sorted ones. A second while loop is then utilized to gather any remaining values that have not yet been placed in mlist.

After this, mlist is returned to sort_split() and the process continues for every recursive level. Once sorted, sort_split returns mlist to mergesort(), allowing mergesort() to then return the sorted mlist to its caller.

Issues & Approaches

I chose this method for a few reasons. My initial approach was to utilize an iterative approach, but I struggled to find a successful solution for properly separating the list. During my struggles, I thought back to my binary search implementation from Lab 2 and was reminded of how easy it was to traverse a list with my recursion and decided to take that approach.

A few issues I ran into involved the loop structure utilized for comparing values in the list. I initially tried to do this with a for loop, but quickly found that keeping track of the three indices used for comparing values was a difficult process without the utilization of multiple or nested for loops. After this realization, I decided to change the loop structure to a while loop. With the while loop, I was able to better keep track of the three indices, each of which accessing their own sub-list, without increasing the time complexity of the subroutine.

Once the while loop structure was flushed, I found that my outputs weren't returning as expected. After running the debugger, I found that if my left and right sections weren't of the same size, then some values were never placed back in the list, resulting in duplicated values that weren't duplicates in the original list and deleted/lost values. To fix this, I implemented a second while-loop after the first that checks if the two indices used to access the left and right sections were less than their respective section's length. If so, then those remainder values are placed at the tail of the list to be returned.

Complexity

The complexity of my algorithm is O(nlog(n)). sort_split() recursively divides the list in half until the halves reach a length of 1, making the subroutine O(log(n)). Then, sort_comparison() and seperate_list() are both O(n) since both rely on the length of the list to perform their functions. Meaning, sort_comparison() eventually compares every element of the input list, and separate_list() works with every element of the input list at least once while using slicing to create two sublist. Resulting in O(n+n(log(n)) = O(2nlog(n)). After dropping constants, we are left with O(nlog(n)).