Austin Youngren

Lab 2: Selecting a Range Reflection – Literacy & Reflection

CS-320-001

Feb. 9, 2024

Approach:

For my approach, I chose to use a binary search that utilizes recursion to search for the requested bounds in a sorted list, then return the values of the requested bounds as a sub-list. I chose this method as I found the for loop implementation difficult to implement while keeping the algorithm at a time complexity, in the worst-case, at O(k + log(n)) or less.

Subroutines:

In my implementation, I created two subroutines, get_lo() and get_hi() which are the heart of the algorithm. Each subroutine takes five parameters, list_s, lo_i, mid_i, hi_i, and the respective boundary they are tasked to find (lo or hi). The parameters lo_i, mid_i, and hi_i act as index pointers for list_s during the binary search and conditional evaluations. These subroutines take five parameters due to their recursive nature.

One of my previous implementations only took three parameters, but I found issues in keeping track of, or the overwriting of, the index pointers as the recursive process continued, creating bugs that caused unexpected sub-lists to be returned during normal functionality cases. Having these functions as separate subroutines made the algorithm easier to debug since I could pinpoint if the issues were happening when the algorithm was finding the lower bound, or when finding the upper bound. Additionally, having them as separate entities ensured I did not add bugs to one while fixing the other.

The subroutine get_lo() first checks if lo_i is greater-than or equal-to hi_i, which serves as the base case for the recursive process. Else, if the value of list_s at index mid_i is greater than or equal to lo, hi_i is assigned the value of mid_i, and mid_i is recalculated using the equation (lo_i + hi_i) divided by 2, using integer division. Then, lo_i is assigned to the recursive call to get_lo(). The Else condition covers the case where the value of list_s, at index mid_i, is less than lo. In this case, lo_i is set to mid_i + 1, mid_i is recalculated in the same manner as the prior step, then lo_i is assigned to the recursive call to get_lo(). The subroutine then returns the index of the lowest value, within the requested bounds, of list_s as lo_i.

The subroutine get_hi(), works in a similar fashion as get_lo(), and utilizes the same base case. The only difference lies in the recursive method being called and the else-if conditional. The else-if checks if hi is less-than the value of list_s at index mid_i, if so, the same actions are taken as the else-if in get_lo(), but lo_i is assigned to the recursive call of get_hi. The else would then cover the condition of hi being greater-than or equal-to list_s[mid_i]. Again taking the same actions as the else in get_lo(), with the exception of the recursive call to get_hi(). This small change in the conditionals ensured that the index after the highest value, within the bound, is returned as lo_i; ensuring the highest value is inclusive in the sub-list.

<u>Extract routine:</u>

The extract(list_s, lo, hi) functionality starts by checking the simple edge cases such as checking if the list is empty or None. After this check, the list length and lower, middle, and higher indices are gathered for list_s; and higher-level edge cases are checked. These special cases check if lo and/or hi contain the value None. If both hi and lo are None, list_s is returned. If lo is None, the algorithm searches for the upper bound and assigns that value to hi_pointer and returns list_s[0: hi_pointer]. If hi is None, the algorithm searches for the lower bound and assigns the value to lo_pointer and returns list_s[lo_pointer:]. The final special case ensures that the value of lo is not greater than hi; if it is, extract() returns None.

If the algorithm passes all of the edge cases, we first search for the upper bound of the requested range and assign it to hi_pointer with subroutine get_hi(). We then search for the lower bound with the get_lo() subroutine. The get_lo() return value is assigned to the variable lo_pointer. Finally, list_s[lo_pointer: hi_pointer], which contains all values within the requested range, is returned from extract()

<u>Complexity:</u>

My algorithm implementation has a worst-case time complexity of O(k + log(n)), where n is the number of values in list_s and k is the number of values in the requested range. This is due to the binary search cutting list_s in half through each recursive call to find a bound of the range within the list. To find both bounds, making this section O(2log(n)). After the recursive calls, we then return the range of values within list_s using slicing, which is O(k). Making the worst- case time complexity O(k + log(n)) after the removal of constants.

<u>Major Implementation Issue:</u>

One of the main issues that plagued me through all versions of my code was finding the upper bound. My returned sub-list would vary in the inclusion of the highest value within the requested range; sometimes it was there and sometimes not. I found that this issue stemmed from me being too focused on the hi_i (high index) variable as the value being returned from the subroutine, get_hi(). This issue caused me to try many different types of index incrementations for lo_i, mid_i, and hi_i out of desperation. I later solved my issue after many hand traces of varied types of sorted lists and found that lo_i should be returned rather than hi_i.