2025-11-01
Eric Petersen

LAB 07: SUB-LIST SORT DESIGN

Step 1: By Hand

```
Step 1:  08  07  02  00  10  17  12  06  18  05  19  04  14  13  03  01  16  11  15  09
Groups: [08][07][02][00  10  17][12][06  18][05  19][04][14][13][03][01  16][11  15][09]
Merged: [07  08][00  02  10  17][06  12  18][04  05  19][13  14][01  03  16][09  11  15]

Step 2:  07  08  00  02  10  17  06  12  18  04  05  19  13  14  01  03  16  09  11  15
Groups: [07  08][00  02  10  17][06  12  18][04  05  19][13  14][01  03  16][09  11  15]
Merged: [00  02  07  08  10  17][04  05  06  12  18  19][01  03  13  14  16][09  11  15]

Step 3:  00  02  07  08  10  17  04  05  06  12  18  19  01  03  13  14  16  09  11  15
Groups: [00  02  07  08  10  17][04  05  06  12  18  19][01  03  13  14  16][09  11  15]
Merged: [00  02  04  05  06  07  08  10  12  17  18  19][01  03  09  11  13  14  15  16]

Step 4:  00  02  04  05  06  07  08  10  12  17  18  19  01  03  09  11  13  14  15  16
Groups: [00  02  04  05  06  07  08  10  12  17  18  19][01  03  09  11  13  14  15  16]
Merged:  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19

Sorted:  00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19
```

Step 2: Approach

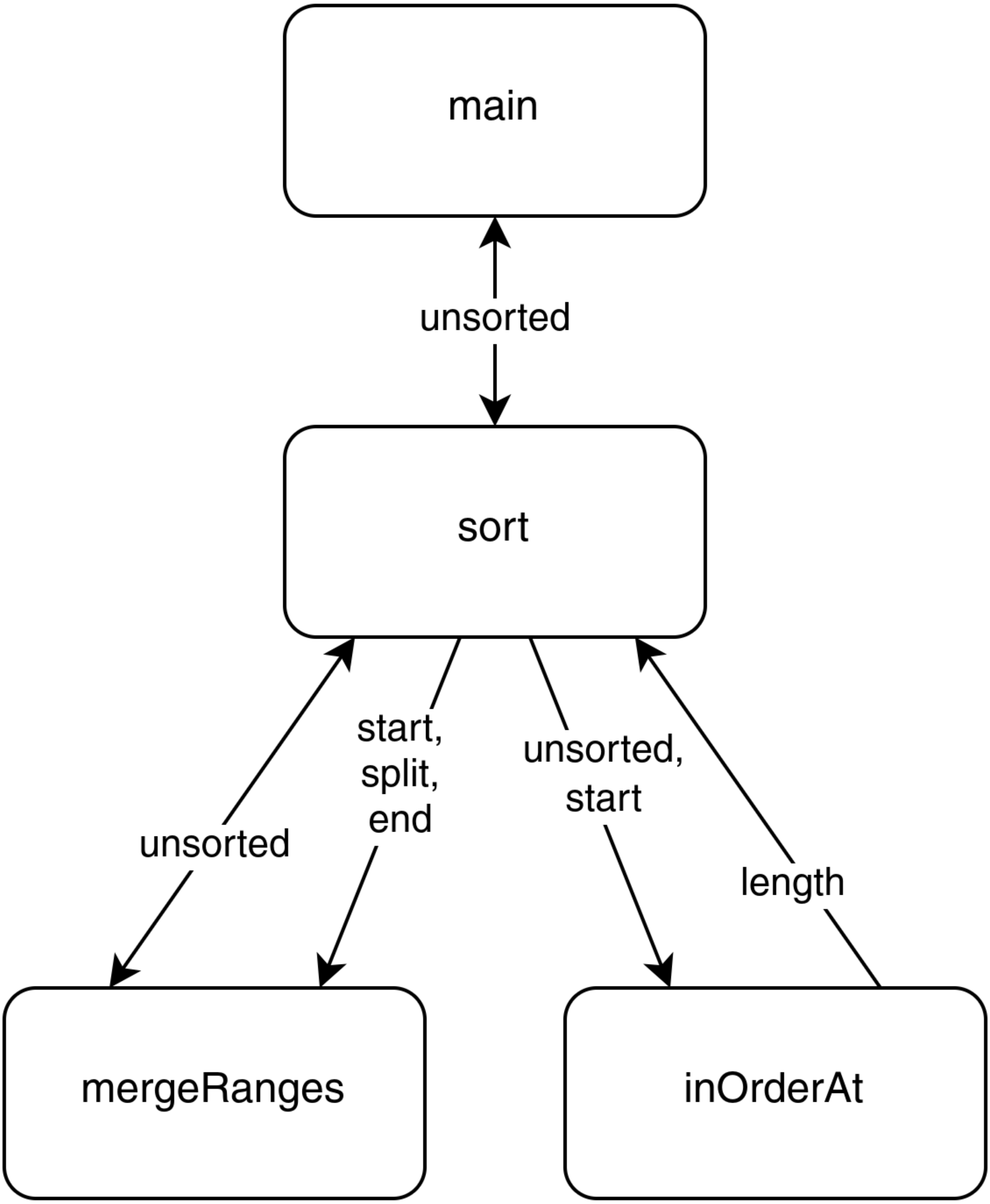
This sorting algorithm consists of three functions; one function to merge two (pre-sorted) ranges of an input list, one function to return the length of the sorted range of an input list at a particular index, and the last function to use the first two functions to sort the list. The process of the third function infinitely repeats a reduction step, until an ending case is recognized, at which point the (now sorted) list is returned. The ending case is true if and only if the length of the in-order list at index 0 is the same as the length of the unsorted list itself; that is, the list is already sorted. The reduction step to this iterative algorithm gets the length of consecutive pairs of in-order ranges and merges them (using the other functions).

Step 3: Pseudocode and Structure Chart

```
FUNCTION mergeRanges(unsorted, start, split, end)
    // Given two consecutive sorted ranges into an unsorted list, this
    // function merges the two ranges to produce one larger sorted range.

    // If the length of one of the ranges is zero, the ranges are already merged
    // ASSERT that end >= split AND split >= start
    IF start == split OR split == end THEN
        RETURN
    END IF

    left_list <- COPY unsorted[start TO split EXCLUSIVE]
    right_list <- COPY unsorted[split TO end EXCLUSIVE]

    left <- 0  // index into left_list
    right <- 0  // index into right_list
    index <- start  // index into unsorted

    // Merge the two lists until one is exhausted
    WHILE left < LENGTH(left_list) AND right < LENGTH(right_list) DO
        // Using <= here for a stable sort, even though it really doesn't matter
        IF left_list[left] <= right_list[right] THEN
            unsorted[index] <- left_list[left]
            left <- left + 1
```

```
        ELSE
            unsorted[index] <- right_list[right]
            right <- right + 1
        END IF
        index <- index + 1
    END WHILE

    // Copy any remaining elements from the right list
    WHILE right < LENGTH(right_list) DO
        unsorted[index] <- right_list[right]
        index <- index + 1
        right <- right + 1
    END WHILE

    // Copy any remaining elements from the left list
    WHILE left < LENGTH(left_list) DO
        unsorted[index] <- left_list[left]
        index <- index + 1
        left <- left + 1
    END WHILE

END FUNCTION


FUNCTION inOrderAt(unsorted, start)
    // Returns the length of the in-order sublist at the start index
    FOR index FROM start + 1 TO LENGTH(unsorted) - 1 DO
        IF unsorted[index] < unsorted[index - 1] THEN
            RETURN index - start
        END IF
    END FOR
    RETURN LENGTH(unsorted) - start
END FUNCTION


FUNCTION sort(unsorted)
    WHILE TRUE DO

        // End condition - the list is already sorted
        IF inOrderAt(unsorted, 0) == LENGTH(unsorted) THEN
            RETURN
        END IF

        // Reduction step - merge consecutive pairs of in-order segments
        start <- 0
        WHILE start < LENGTH(unsorted) DO
            split <- start + inOrderAt(unsorted, start)
            end <- split + inOrderAt(unsorted, split)
            mergeRanges(unsorted, start, split, end)
            start <- end
        END WHILE

    END WHILE
END FUNCTION
```

Please note that for my structure chart, I used double-arrowed lines to mark parameters which are not strictly returned, but which are mutated. This was according to the recommendation of a comment on stack overflow, because I could not find how to represent this in the book.



Step 4: Microsoft Copilot

```
    Please give me a **pseudocode solution** to the following algorithm approach:
This sorting algorithm consists of three functions; one function to merge two (pre-
sorted) ranges of an input list, one function to return the length of the sorted
range of an input list at a particular index, and the last function to use the
first two functions to sort the list. The process of the third function infinitely
repeats a reduction step, until an ending case is recognized, at which point the
```

(now sorted) list is returned. The ending case is true if and only if the length of the in-order list at index 0 is the same as the length of the unsorted list itself; that is, the list is already sorted. The reduction step to this iterative algorithm gets the length of consecutive pairs of in-order ranges and merges them (using the other functions).

```
function merge_ranges(list, start1, len1, start2, len2):
    # Merge two sorted ranges: list[start1:start1+len1] and list[start2:start2+len2]
    merged = []
    i = start1
    j = start2
    end1 = start1 + len1
    end2 = start2 + len2

    while i < end1 and j < end2:
        if list[i] <= list[j]:
            merged.append(list[i])
            i += 1
        else:
            merged.append(list[j])
            j += 1

    # Append remaining elements
    while i < end1:
        merged.append(list[i])
        i += 1
    while j < end2:
        merged.append(list[j])
        j += 1

    # Replace original ranges with merged result
    for k in range(len(merged)):
        list[start1 + k] = merged[k]

    return list


function sorted_range_length(list, index):
    # Return length of sorted (non-decreasing) range starting at index
    if index >= length(list):
        return 0

    length = 1
    while index + length < length(list) and list[index + length - 1] <= list[index + length]:
        length += 1

    return length


function iterative_sort(list):
    while true:
        # Check ending condition
        if sorted_range_length(list, 0) == length(list):
            return list  # Fully sorted

        index = 0
        while index < length(list):
            len1 = sorted_range_length(list, index)
            len2 = sorted_range_length(list, index + len1)

            if len2 > 0:
                list = merge_ranges(list, index, len1, index + len1, len2)
                index += len1 + len2
            else:
                index += len1

    return list
```

Step 5: Compare and Contrast

Both my pseudocode solution, and that of the LLM contain the requisite number of functions. Both appear to to have the same structure for the sort function. My pseudocode attempts to copy the left and right sublists and update the original list based off of these sublists for the merge function. The LLM solution instead builds up the merged sublists from the original, then patches the original after the sublists are merged. The sorted range length function is logically equivalent, but feels to be cleaner in my pseudocode. The LLM does not call the merge function if the length of one or both sublist lengths are zero, while my merge function exists early if one or both sublist lengths are zero.

Looking at the solution that Microsoft Copilot generated, my understanding of the algorithm described in step 1 is justified. Both algorithms in step 3 and step 4 are semantically equivalent, with the most notable difference being in how the merge function handles the merged sublists. Because my merge function copies two sublists at the start of the function, and the LLM code appends to a new list for every single element in each of the sublists, I would say that my solution is more optimal than that of the LLM.

Step 6: Update

The one possible improvement I can see for my pseudocode would involve **where** to place the check for if I can skip merging two sections (whether or not one or both lengths are 0). I can either place the check at the function call site, or in the function itself. The LLM placed this check at the call site, and I have placed the check at the beginning of my merge function. Given that the check would need to be duplicated at each call-site to be effective - and that I do not expect this function to be used in situations where people routinely assert that the length of each sublist is zero, I will maintain the current design and placement of my pseudocode. I believe that I have the best possible design.