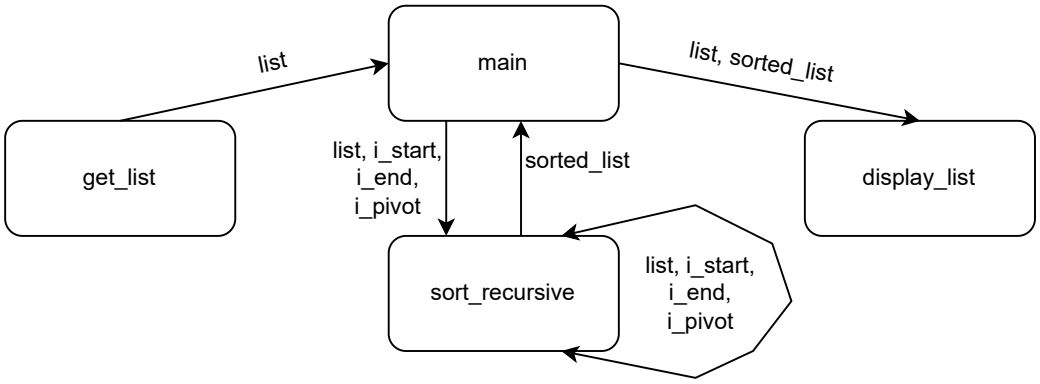

main

Cohesion: The cohesion is strong because it directs the program, initiating the various functions as needed. No extra features are present.

Coupling: Main receives no variables, nor returns any, so it is trivial.

get_list

Cohesion: This function gets the list from the user (or file, if testing), then returns it. With no extra features, the coupling is strong.

Coupling: The coupling here is simple. The inputs from the user are easy to make sure they are numbers, and beyond that, the end product is an easy to use list.

sort_recursive

Cohesion: This function takes the list from before, uses the sorting algorithm, then splits the list and each half keeps sorting and splitting until the whole list is sorted. But there are no more features here than are required for the program, so the cohesion is strong.

Coupling: The coupling here is encapsulated. All the verification of information from the user was completed in get_input(), so it is guaranteed to be in a valid state by the time the data reaches this function.

display_list

Cohesion: The cohesion is strong because the function takes the provided values and sorts them, and does nothing extra.

Coupling: The coupling here is simple because the list is easy to interpret and pull values out of.

```
main()
    // First, get the list.
    // Overall effeciency: O(log n)
    list <- get_list() // A O(1)

    // Next, set up parameters for the sort function.
    i_start <- 0 // B O(1)
    i_end <- len(list) - 1 // B O(1)
    i_pivot <- ((i_end - i_start) / 2).roundup() // B O(1)

    // Sort the list using the above parameters.
    sorted_list <- sort_recursive(list, i_start, i_end, i_pivot) // O(1)

    // Lastly, print the list.
    display_list(list, sorted_list) // O O(1)

sort_recursive(list, i_start, i_end, i_pivot)
    // Sort the list, each time dividing and creating a recursive loop
until the list is all sorted.

    // If the sublist is less than 1 item long, you don't need to sort it
anymore.
    IF len(list) < 2
        return list

    // If the sublist is only 2 items, compare and sort them.
    ELIF len(list) == 2
        IF list[0] > len[1]
            value_pivot <- list[1] // C O(1)
            list[1] <- list[0] // C O(1)
            list[0] <- value_pivot // C O(1)
        return list

    // Otherwise, split the list using the algorithm and sort it.
    ELSE

        // Set up listeners for when to switch values.
        left_switch_point_found <- False // D O(1)
        right_switch_point_found <- False // D O(1)
        list_sorted <- True // D O(1)

        // Set the pivot point's value so the program doesn't have to look
it up everytime.
        value_pivot <- list[i_pivot] // E O(1)
```

```
        // Set the begin and end points before i_start and i_end get moved
for ease of dividing the list later.
        i_start_static <- i_start // E O(1)
        i_end_static <- i_end // E O(1)

        // Keep sorting, moving towards the middle until the program
reaches the mid point.
        UNTIL i_start == i_pivot AND i_end == i_pivot // O(n)

            // If the number on the left of the pivot is greater than the
pivot value, mark it for swapping.
            IF list[i_start] > value_pivot
                i_left_switch_point <- i_start // F O(1)
                left_switch_point_found <- True // F O(1)
                list_sorted <- False // F O(1)

            // If the i_start reaches the pivot point without finding an
available value, set it to swap with whatever the right side finds.
            ELIF i_start == i_pivot:
                i_left_switch_point <- value_pivot // G O(1)
                left_switch_point_found <- True // G O(1)
            ELSE:
                i_start++ // H O(1)

            // Do the same with the right side, but if the values are
lower.
            IF list[i_end] < value_pivot
                i_right_switch_point <- i_end // I O(1)
                right_switch_point_found <- True // I O(1)
                list_sorted <- False // I O(1)

            // If the i_end reaches the pivot point without finding an
available value, set it to swap with whatever the left side finds.
            ELIF i_start == i_pivot:
                i_left_switch_point <- value_pivot // J O(1)
                left_switch_point_found <- True // J O(1)

            ELSE:
                i_end-- // K O(1)

            // If the points are ready to switch, switch them.
            IF left_switch_point_found == True and
right_switch_point_found == True // O(1)

                // Assign values.
```

```
                left_switch_value <- list[i_left_switch_point] // L O(1)
                right_switch_value <- list[i_right_switch_point] // L O(1)

                // Reassign values within the list.
                list[i_left_switch_point] <- right_switch_value // M O(1)
                list[i_right_switch_point] <- left_switch_value // M O(1)

                // Reset listeners for next swap.
                left_switch_point_found <- False // N O(1)
                right_switch_point_found <- False // N O(1)

        // If list had to switch values, it is not sorted, so set up
recursive loop that
        // will sort half of the list until the whole list is sorted.
        IF list_sorted == False

            // Remember, the format for the function call is:
sort_recursive(list, i_start, i_end, i_pivot)
            list <- sort_recursive(list, i_start_static, i_pivot,
(i_pivot)/2.roundup()) // Left half O(log n)
            list <- sort_recursive(list, i_pivot + 1, i_end_static,
(i_pivot*1.5).roundup()) //  Right half O(log n)

    // Finally, return the sorted list.
    return list

display_list(list, sorted_list)
    // Display the list, then the sorted list.
    PUT list // O(n)
    PUT sorted_list // O(n)
```

Name	Description	Input	Output
Empty List	Program should have no problem sorting no items.	[]	[]
Single Item	If the list is 1 item, it is already sorted.	[5]	[5]
Short Unsorted List	The program should be able to sort 2 unsorted items	[5, 3]	[3, 5]
Short Sorted List	Same as before, but should already recognize that it is sorted.	[3, 5]	[3, 5]
Reverse order w/ odd items	If the list is backwards, still sort it.	[5, 4, 3, 2, 1]	[1, 2, 3, 4, 5]
Mixed-up w/ odd items	If the list is all mixed up, still sort it.	[2, 3, 4, 1, 5]	[1, 2, 3, 4, 5]
Sorted with odd items	Like short sorted list, but still recognizable as sorted.	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
Reverse order w/ even items	Like reverse order with odd, but program still works with even numbers	[4, 3, 2, 1]	[1, 2, 3, 4]
Mixed-up w/ even items	^^	[3, 1, 2, 4]	[1, 2, 3, 4]
Sorted with even items	^^	[1, 2, 3, 4]	[1, 2, 3, 4]
Duplicates	The list will still sort the list, even if there are duplicates.	[1, 3, 4, 3, 2]	[1, 2, 3, 3, 4]

	Test Case: Short Unsorted List					
	list	sorted_list	i_start	i_pivot	i_end	value_pivot
A	[5, 3]	/	/	/	/	/
B	[5, 3]	/	0	1	1	/
C	[3, 5]	/	0	1	1	3
O	[5, 3]	[3, 5]	0	1	1	3

	list	sorted_list	i_start	i_pivot	i_end	value_pivot	left_switch_point_found
A	[2, 3, 4, 1, 5]	/	/	/	/	/	/
B	[2, 3, 4, 1, 5]	/	0	2	4	/	/
D	[2, 3, 4, 1, 5]	/	0	2	4	/	FALSE
E	[2, 3, 4, 1, 5]	/	0	2	4	4	FALSE
H	[2, 3, 4, 1, 5]	/	1	2	4	4	FALSE
K	[2, 3, 4, 1, 5]	/	1	2	3	4	FALSE
H	[2, 3, 4, 1, 5]	/	2	2	3	4	FALSE
I	[2, 3, 4, 1, 5]	/	2	2	3	4	FALSE
G	[2, 3, 4, 1, 5]	/	2	2	3	4	TRUE
L	[2, 3, 4, 1, 5]	/	2	2	3	4	TRUE
M	[2, 3, 1, 4, 5]	/	2	2	3	4	TRUE
N	[2, 3, 1, 4, 5]	/	2	2	3	4	FALSE
G	[2, 3, 1, 4, 5]	/	2	2	3	4	TRUE
K	[2, 3, 1, 4, 5]	/	2	2	2	4	TRUE
D	[2, 3, 1, 4, 5]	/	0	1	2	/	FALSE
E	[2, 3, 1, 4, 5]	/	0	1	2	3	FALSE
H	[2, 3, 1, 4, 5]	/	1	1	2	3	FALSE
I	[2, 3, 1, 4, 5]	/	1	1	2	3	FALSE
G	[2, 3, 1, 4, 5]	/	1	1	2	3	TRUE
I	[2, 3, 1, 4, 5]	/	1	1	2	3	TRUE
L	[2, 3, 1, 4, 5]	/	1	1	2	3	TRUE
M	[2, 1, 3, 4, 5]	/	1	1	2	3	TRUE
N	[2, 1, 3, 4, 5]	/	1	1	2	3	FALSE
G	[2, 1, 3, 4, 5]	/	1	1	2	3	TRUE
K	[2, 1, 3, 4, 5]	/	1	1	1	3	TRUE
C	[1, 2, 3, 4, 5]	/	0	1	1	1	/
O	[2, 3, 4, 1, 5]	[1, 2, 3, 4, 5]	0	2	4	/	/

Test Case: Mixed-up w/ odd items

right_switch_point_found	i_left_switch_point	i_right_switch_point	left_switch_value
/	/	/	/
/	/	/	/
FALSE	/	/	/
FALSE	/	/	/
FALSE	/	/	/
FALSE	/	/	/
FALSE	/	/	/
TRUE	/	3	/
TRUE	2	3	/
TRUE	2	3	4
TRUE	2	3	4
FALSE	2	3	4
FALSE	2	3	4
FALSE	2	3	1
FALSE	/	/	/
FALSE	/	/	/
FALSE	/	/	/
TRUE	/	2	/
TRUE	1	2	/
TRUE	1	2	/
TRUE	1	2	3
TRUE	1	2	3
FALSE	1	2	3
FALSE	1	2	3
FALSE	1	2	3
/	/	/	/
/	/	/	/

right_switch_value	list_sorted	i_start_static	i_end_static
/	/	/	/
/	/	/	/
/	TRUE	/	/
/	TRUE	0	4
/	TRUE	0	4
/	TRUE	0	4
/	TRUE	0	4
/	FALSE	0	4
/	FALSE	0	4
1	FALSE	0	4
1	FALSE	0	4
1	FALSE	0	4
1	FALSE	0	4
1	FALSE	0	4
/	TRUE	/	/
/	TRUE	0	2
/	TRUE	0	2
/	FALSE	0	2
/	FALSE	0	2
/	FALSE	0	2
1	FALSE	0	2
1	FALSE	0	2
1	FALSE	0	2
1	FALSE	0	2
1	FALSE	0	2
/	/	/	/
/	/	/	/