1. We use a whitespace-separated parenthesized list to denote a list.

Call	Return
`merge((1 2 3 4 5), (2 4 6 8 10))`	`(1 2 2 3 4 4 5 6 8 10)`
`merge((2 3 4 5), (2 4 6 8 10))`	`(2 2 3 4 4 5 6 8 10)`
`merge((3 4 5), (2 4 6 8 10))`	`(2 3 4 4 5 6 8 10)`
`merge((3 4 5), (4 6 8 10))`	`(3 4 4 5 6 8 10)`
`merge((4 5), (4 6 8 10))`	`(4 4 5 6 8 10)`
`merge((5), (4 6 8 10))`	`(4 5 6 8 10)`
`merge((5), (6 8 10))`	`(5 6 8 10)`
`merge(NULL, (6 8 10))`	`(6 8 10)`

2. The following process has the shape of a tree. Drawing this process as a sequence, like that given above, is not easy. Instead, we begin by evaluating `MergeSort((8 7 6 5 4 3 2 1))`, where each call to `MergeSort` becomes a call to `split`, and each `split` becomes a call to

`merge(MergeSort(`⟨*odd-list*⟩`), MergeSort(`⟨*even-list*⟩`))`

A list is split after a single call to `split`, and two lists are merged after a single call to `merge` where there are no function calls in the arguments (i.e. `merge((4 8), (2 6))` gets merged but `merge((4 8), merge((6), (2)))` does not). The exercise is ambiguous (to me) so we draw the process in this way for now.

```
MergeSort((8 7 6 5 4 3 2 1))
split((8 7 6 5 4 3 2 1))
merge(MergeSort((8 6 4 2)), MergeSort((7 5 3 1)))
merge(split((8 6 4 2)), MergeSort((7 5 3 1)))
merge(merge(MergeSort((8 4)), MergeSort((6 2))), MergeSort((7 5 3 1)))
merge(merge(split((8 4)), MergeSort((6 2))), MergeSort((7 5 3 1)))
merge(merge(merge(MergeSort((8)), MergeSort((4))), MergeSort((6 2))), MergeSort((7 5 3 1)))
merge(merge(merge((8), (4)), MergeSort((6 2))), MergeSort((7 5 3 1)))
merge(merge((4 8), MergeSort((6 2))), MergeSort((7 5 3 1)))
merge(merge((4 8), split((6 2))), MergeSort((7 5 3 1)))
merge(merge((4 8), merge(MergeSort((6)), MergeSort((2)))), MergeSort((7 5 3 1)))
merge(merge((4 8), merge((6), (2))), MergeSort((7 5 3 1)))
merge(merge((4 8), (2 6)), MergeSort((7 5 3 1)))
merge((2 4 6 8), MergeSort((7 5 3 1)))
merge((2 4 6 8), split((7 5 3 1)))
merge((2 4 6 8), merge(MergeSort((7 3)), MergeSort((5 1))))
merge((2 4 6 8), merge(split((7 3)), MergeSort((5 1))))
merge((2 4 6 8), merge(merge(MergeSort((7)), MergeSort((3))), MergeSort((5 1))))
merge((2 4 6 8), merge(merge((7), (3)), MergeSort((5 1))))
merge((2 4 6 8), merge((7 3), MergeSort((5 1))))
merge((2 4 6 8), merge((7 3), split((5 1))))
merge((2 4 6 8), merge((7 3), merge(MergeSort((5)), MergeSort((1)))))
merge((2 4 6 8), merge((7 3), merge((5), (1))))
merge((2 4 6 8), merge((7 3), (1 5)))
merge((2 4 6 8), (1 3 5 7))
(1 2 3 4 5 6 7 8)
```

3.

```
int length(LIST list)
{
    if (list == NULL) return 0;
    else return 1 + length(list->next);
}

LIST split(LIST list, int n)
{
    LIST rest;

    if (n == 0) {
        rest = list->next;
        list->next = NULL;
        return rest;
    }
    else return split(list->next, n-1);
}

LIST MergeSort(LIST list)
{
    LIST SecondList, ThirdList;
    int len;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return list;
    else {
        len = length(list);
        ThirdList = split(list, 2 * len / 3);
        SecondList = split(list, len / 3);
        return merge(MergeSort(list),
                     merge(MergeSort(SecondList),
                           MergeSort(ThirdList)));
    }
}
```