CSC148 - Recursive Sorting Algorithm Efficiency

Now that we've studied both mergesort and quicksort, we'll take a look at the running times of these algorithms. To analyse the running time of recursive algorithms, we need to study two things:

- the running time of the non-recursive operations in the code
- the structure of the recursion, i.e., how many recursive calls we make, and on what "smaller" inputs

This is a new technique that we have not covered before, so this worksheet will focus on getting the pieces ready to do a complete analysis together in lecture.

1. Here is the implementation of the helper function `_partition` used by quicksort.

```
def _partition(lst: List, pivot: Any) -> Tuple[List, List]:
    smaller, bigger = [], []
    for item in lst:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)
    return smaller, bigger
```

n iterations

constant

What is the Big-Oh running time of `_partition` in terms of n, the length of its input list?

$O(n)$

2. How would your answer change if each item were inserted at the *front* of the relevant list instead, for example: `smaller.insert(0, item))`?

$O(n^2)$

3. Here is the implementation of the helper function `_merge` used by mergesort.

```
def _merge(lst1: List, lst2: List) -> List:
    index1, index2 = 0, 0
    merged = []
    while index1 < len(lst1) and index2 < len(lst2):
        if lst1[index1] <= lst2[index2]:
            merged.append(lst1[index1])
            index1 += 1
        else:
            merged.append(lst2[index2])
            index2 += 1

    return merged + lst1[index1:] + lst2[index2:]
```

# of comparisons (# of loop iterations)

L1 1 2 3 4
L2 5 6 7 8

1 2 3 4 5 6 7 8

L1 1 3 5 7
L2 2 4 6 8

1 2 3 4 5 6 7 8

Let n_1 be the length of `lst1` and n_2 be the length of `lst2`. (Note that `_merge` *does* work correctly even when given lists of different lengths.) What is the maximum number of loop iterations for `_merge`, in terms of n_1 and/or n_2?

$n1 + n2 \rightarrow O(\underbrace{n1 + n2}_{n})$ if $n = n1 + n2$, this is $O(n)$

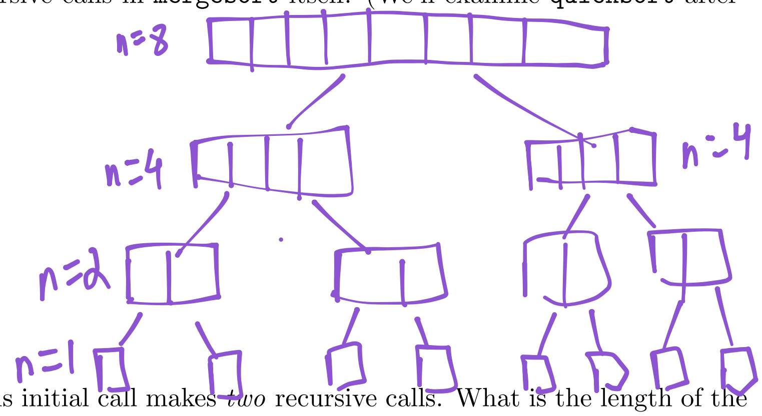
4. Now consider the loop body. Does any operation in the loop body have a running time that depends on n_1 and/or n_2? You should be able to justify your answer for every single operation given in this code, from `<=` to `merged.append(...)` to `lst1[index1]`.

lst1[index1:] will take n1 - index1 steps
lst2[index2:] will take n2 - index2 steps

5. Now we'll turn our attention to the structure of the recursive calls in `mergesort` itself. (We'll examine `quicksort` afterwards.)

```python
def mergesort(lst: List) -> List:
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        return _merge(left_sorted, right_sorted)
```



(a) Suppose we call `mergesort` on a list of length **8**. This initial call makes *two* recursive calls. What is the length of the input lists to these two calls?

4

(b) Each of the recursive calls in part (a) makes two new recursive calls. What is the length of the input lists for these new recursive calls, and how many of them are there in total?

len 2, 4 in total

(c) Finally, each of the recursive calls in part (b) makes two new recursive calls. What is the length of the input lists for these new recursive calls, and how many of them are there in total?

len 1, 8 in total

(d) To make the pattern clear, fill out the table below. We've filled out the first column for you (showing the initial call); complete the rest of the table.

Input list length	8	4	2	1
Number of `mergesort` calls on lists of that length	1	2	4	8

* Based on all the above, what is the overall big O running time of mergesort?

6. Now consider the quicksort algorithm.

```python
def quicksort(lst: List) -> List:
    if len(lst) < 2:
        return lst[:]
    else:
        pivot = lst[0]
        smaller, bigger = _partition(lst[1:], pivot)
        return quicksort(smaller) + [pivot] + quicksort(bigger)
```

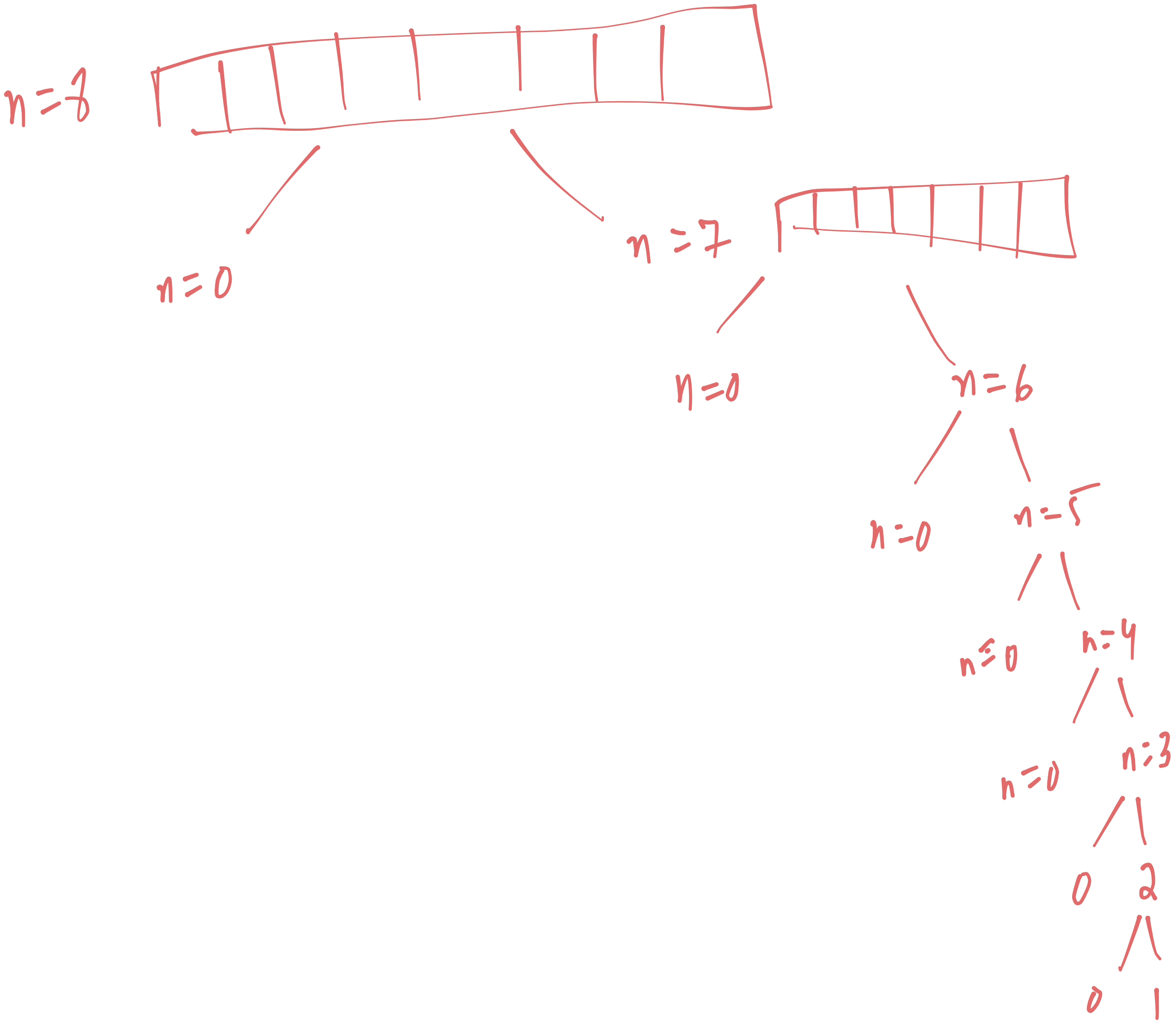
Its recursive step also makes two recursive calls, but unlike mergesort, the input list lengths are not necessarily always the same size.

(a) Suppose we call `quicksort([0, 10, 20, 30, 40, 50, 60, 70])`. After the `_partition` line, what are the lengths of `smaller` and `bigger`?

0 7

(b) Now let's look at the recursive structure of this call. Assume that for *every* recursive call to `quicksort`, the first element of the input is always the smallest element of the list. Using this assumption, complete the following table, which shows the input list length vs. number of `quicksort` calls. (You'll need to add more columns.)

Input list length	8	7	6	5	4	3	2	1	0
Number of `quicksort` calls on lists of that length	1	1	1	1	1	1	1	1	~~8~~ 7



n=8
n=0
n=7
n=0
n=6
n=0
n=5
n=0
n=4
n=0
n=3
0
2
0
1