# CSC111 Lecture 20: Recursive Sorting Algorithms, Part 2

Hisbaan Noorani

March 24, 2021

## Contents



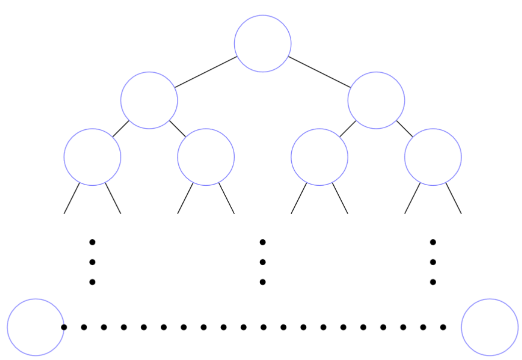
## 1 Exercise 1: Running-time analysis for mergesort

Here is our mergesort implementation.

```python
def mergesort(lst: list) -> list:
    if len(lst) < 2:
        return lst.copy()  # Use the list.copy method to return a new list object
    else:
        # Divide the list into two parts, and sort them recursively.
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        # Merge the two sorted halves. Using a helper here!
        return _merge(left_sorted, right_sorted)
```

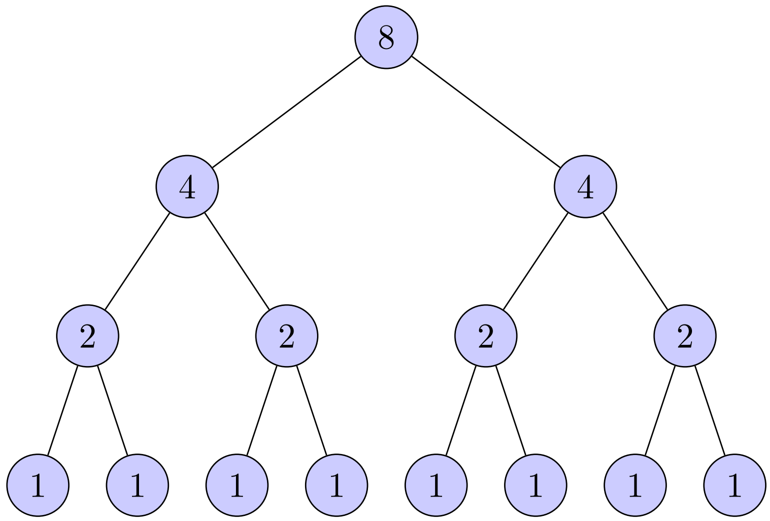
We saw in lecture that the recursive call tree for `mergesort` is a binary tree like the following:



In this exercise, you'll complete a running-time analysis for mergesort.

1. Suppose we call `mergesort` on a list of length `8`. Draw the corresponding recursion diagram, and inside each node write down the non-recursive running time of the call, which we count as equal to the *size of the input list* for that call.

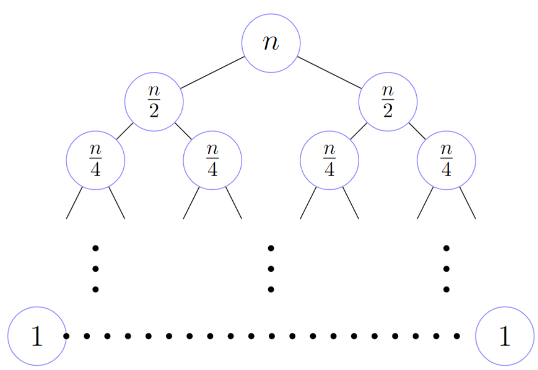
   (For example, the root of the tree should contain an “8”, and its two children should be “4”s.)




2. Compute the total of the numbers in your above diagram. This gives you the total running time for `mergesort` on a list of length 8.

   The total of the numbres in the above diagram is $8 \cdot 1 + 4 \cdot 2 + 2 \cdot 4 + 1 \cdot 8 - 32$.

3. Now suppose we have a list of length $n$, where $n$ is a power of 2. Fill in the recursion diagram below with the corresponding non-recursive running times. The root node should be filled in with $n$.




4. Compute the total of the numbers in your above diagram, again assuming $n$ is a power of 2.

   Hint: consider the sum of the numbers in each *level* of the tree.

   The total of the numbers in the above diagram is $n$ multiplied by the height of the tree which is $\log n$ therefore, it is $n \log n$.

## 2 Exercise 2: Quicksort running time and uneven partitions

Now consider the quicksort algorithm.

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

        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)

        return smaller_sorted + [pivot] + bigger_sorted
```

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

1. Suppose we call `quicksort([0, 10, 20, 30, 40])`. After the `_partition` call, what are `smaller` and `bigger`?

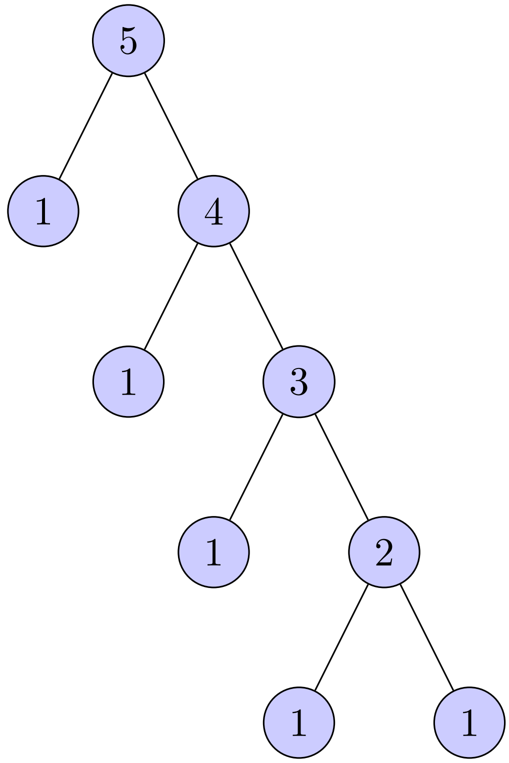
   `smaller = [], bigger = [10, 20, 30, 40]`

2. Draw a recursion tree showing the *inputs* to each recursive call, when we call `quicksort([0, 10, 20, 30, 40])`. We've started the first two levels for you below. The node containing represents a recursive call on an empty list.

```
[0, 10, 20, 30, 40]
 /               \
[]        [10, 20, 30, 40]
           /            \
          []       [20, 30, 40]
                    /        \
                   []      [30, 40]
                            /    \
                           []   [40]
```

3. Now redraw the recursion tree, but with the *non-recursive running time* in each node.

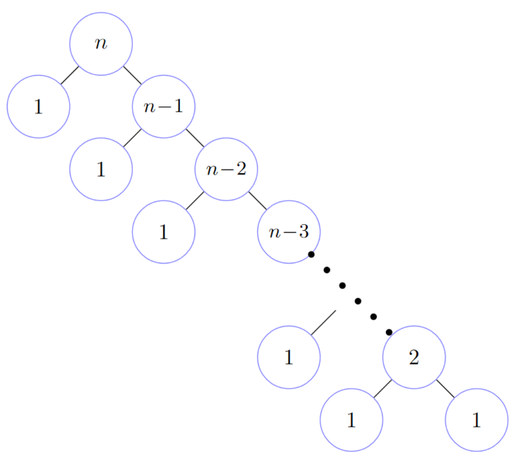
   - For an empty list, the non-recursive running time is 1.
   - For a non-empty list, the non-recursive running time is the *length of the list*.




4. Add up all of the numbers in the above diagram.

   $5 + 1 + 4 + 1 + 3 + 1 + 2 + 1 + 1 = 19$

5. Generalize the above calculation for a call to quicksort with a list of length $n$, when the chosen pivot is always the *smallest* element in the list.




   This leads to a running time of $\Theta(n^2)$

# 3 Additional exercises

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

```python
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
```

(a) Analyse the running time of `_partition` in terms of $n$, the length of its input list.

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

2. Analyse the running time of your `_in_place_partition` implementation from last class.