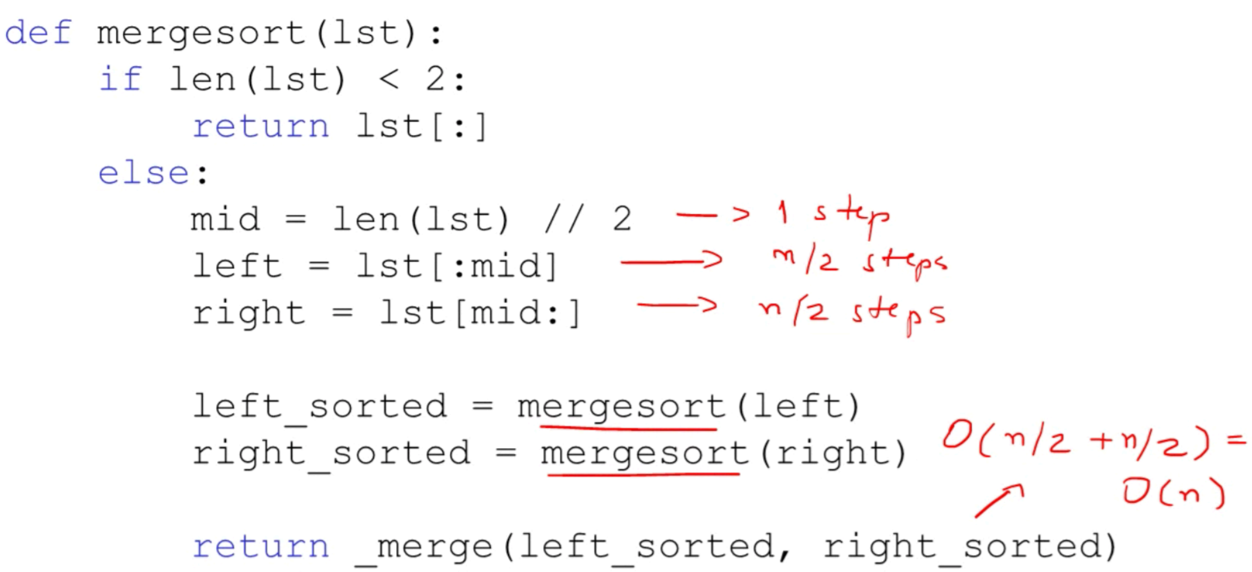
7.2 Efficiency of Recursive Sorting Algorithms

- To analyze the efficiency of recursive functions :
    - Analyse non-recursive part of the code
    - Factor in the cost for each recursive call made

MERGESORT:



```
def mergesort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left = lst[:mid]
        right = lst[mid:]

        left_sorted = mergesort(left)
        right_sorted = mergesort(right)

        return _merge(left_sorted, right_sorted)
```

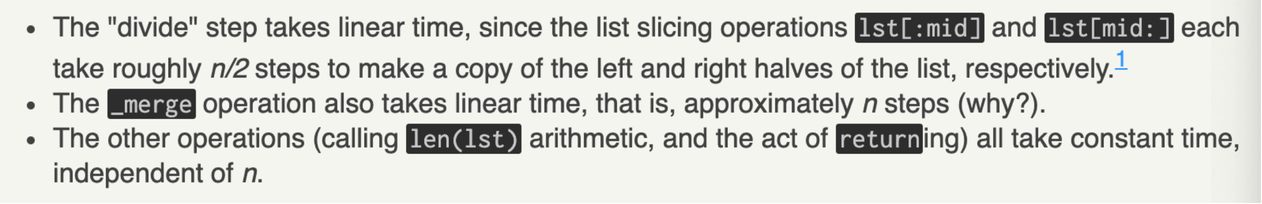
—> 1 step

—> n/2 steps

—> n/2 steps

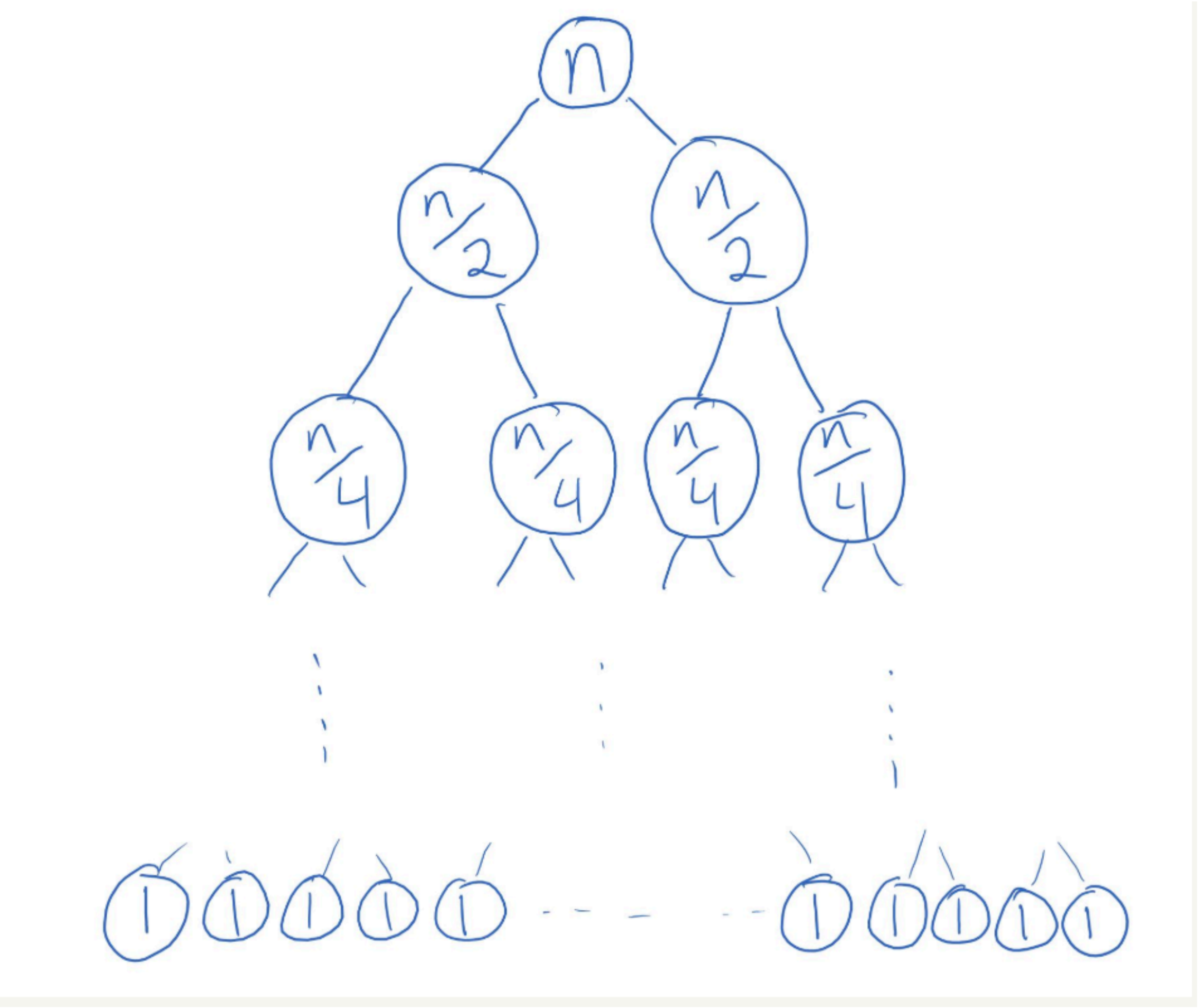
$O(n/2 + n/2) = O(n)$

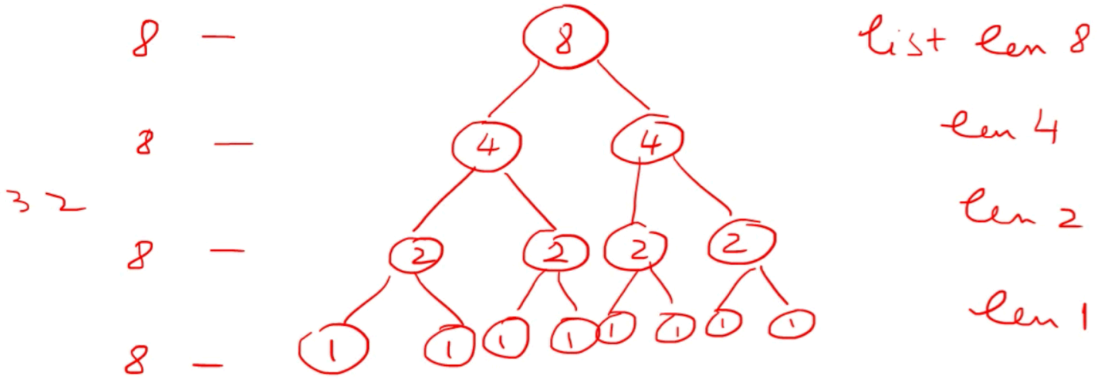
- For a list of length n where n >=2 :
    -



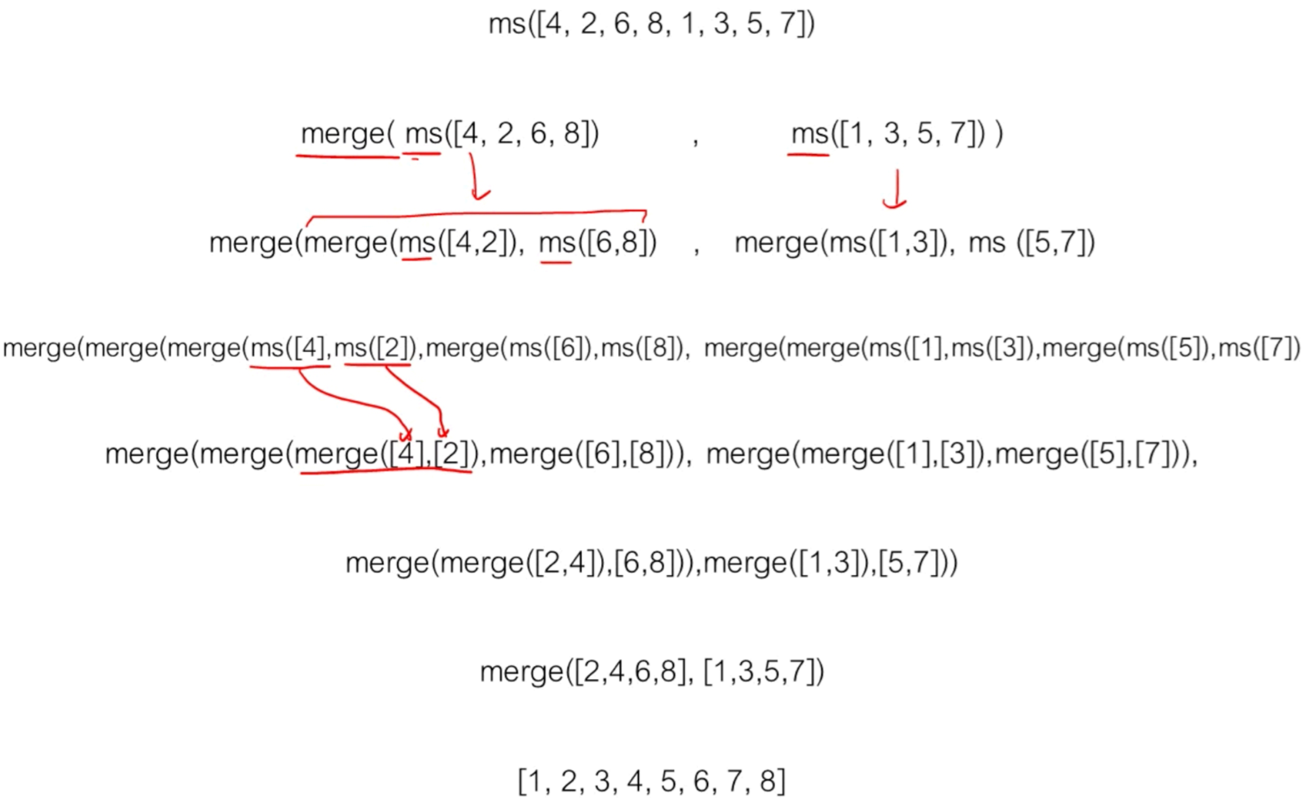
- The "divide" step takes linear time, since the list slicing operations `lst[:mid]` and `lst[mid:]` each take roughly *n/2* steps to make a copy of the left and right halves of the list, respectively.[1]
- The `_merge` operation also takes linear time, that is, approximately *n* steps (why?).
- The other operations (calling `len(lst)` arithmetic, and the act of `return`ing) all take constant time, independent of *n*.

- For the non-recursive part is linear
- Recursive part:





- The height of the tree = recursion depth (# recursive calls made before base case is reached)
- Recursion depth of merge sort:
- # of times n is divided by 2 to get to 1 (2^k = n) -> log n
- MERGESORT Worst-case/best-case running time: O(n log n)



ms([4, 2, 6, 8, 1, 3, 5, 7])

merge(ms([4, 2, 6, 8]) , ms([1, 3, 5, 7]))

merge(merge(ms([4,2]), ms([6,8]) , merge(ms([1,3]), ms ([5,7])

merge(merge(merge(ms([4],ms([2]),merge(ms([6]),ms([8]), merge(merge(ms([1],ms([3]),merge(ms([5]),ms([7])

merge(merge(merge([4],[2]),merge([6],[8])), merge(merge([1],[3]),merge([5],[7])),

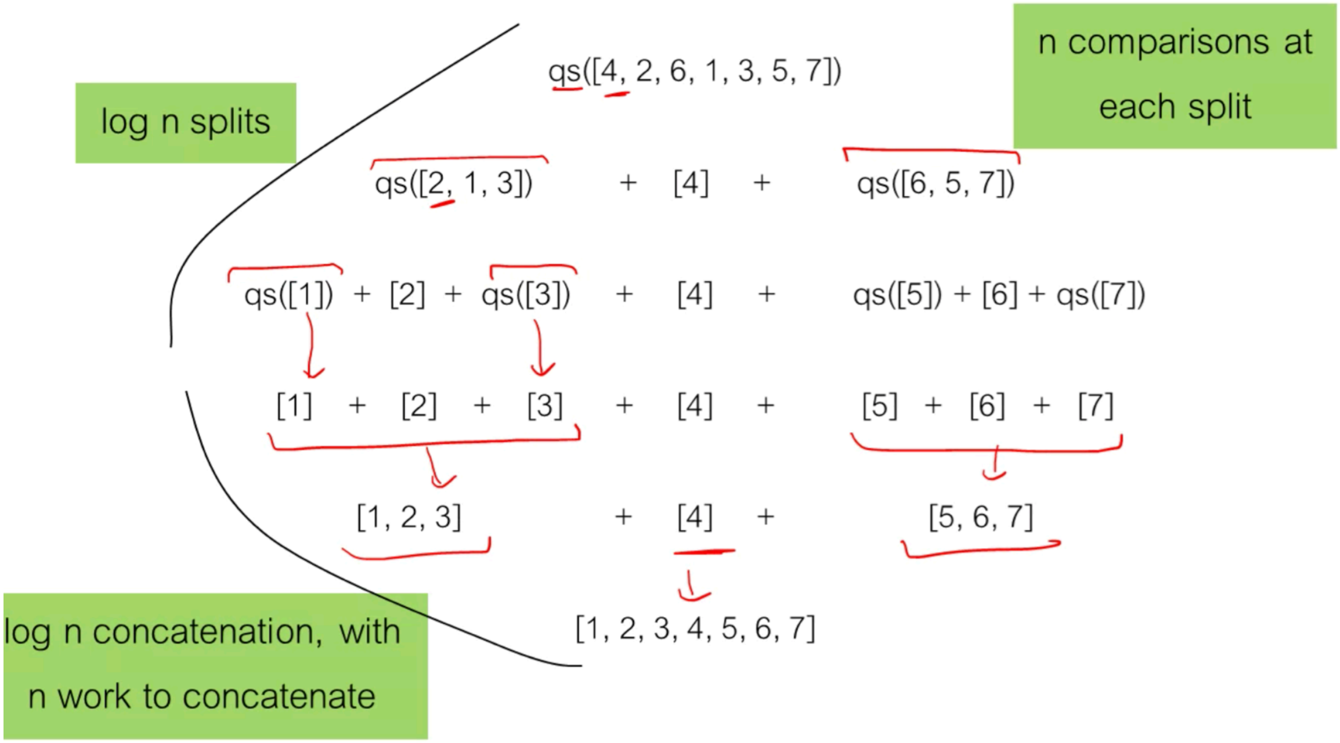
merge(merge([2,4]),[6,8])),merge([1,3]),[5,7]))

merge([2,4,6,8], [1,3,5,7])

[1, 2, 3, 4, 5, 6, 7, 8]

QUICKSORT

- If the pivot is always in the middle then the running time is the same as merge sort!



qs([4, 2, 6, 1, 3, 5, 7])

qs([2, 1, 3]) + [4] + qs([6, 5, 7])

qs([1]) + [2] + qs([3]) + [4] + qs([5]) + [6] + qs([7])

[1] + [2] + [3] + [4] + [5] + [6] + [7]

[1, 2, 3] + [4] + [5, 6, 7]

[1, 2, 3, 4, 5, 6, 7]

```
def quicksort(lst):
   if len(lst) < 2:
      return lst[:]
   else:
      pivot = lst[0]

      smaller, bigger = _partition(lst[1:], pivot)

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

      return smaller_sorted + [pivot] +
             bigger_sorted
```

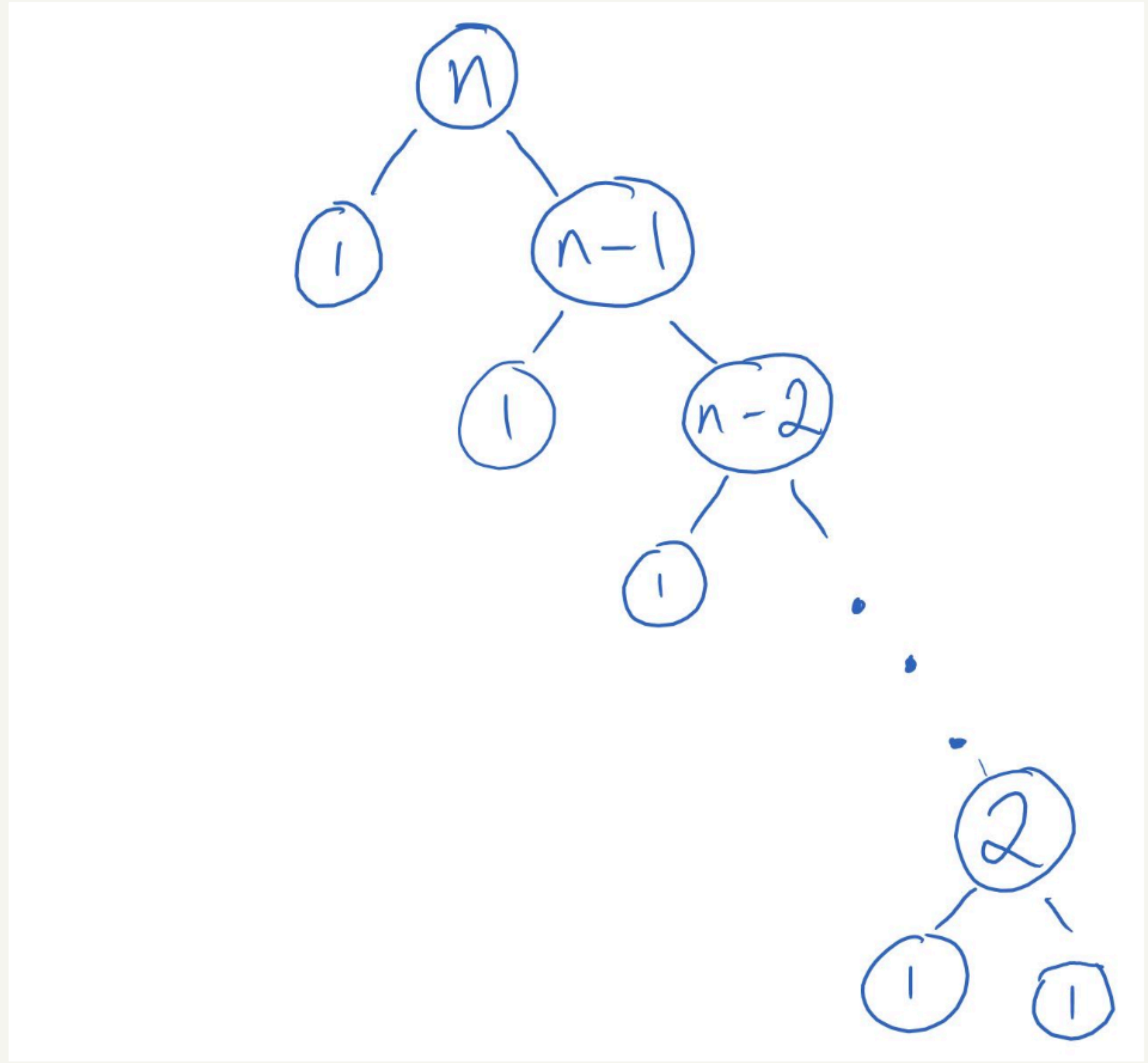
→ 1 step

→ n -1 steps

— n items to copy over to create a new list

n log n

- If the pivot yields uneven partition (one empty, one with the rest) we get:
  - The size decreases by 1 at each recursive call
  - Adding the cost of each level gives this (n^2) expression
  - $(n-1) + [n + (n-1) + (n-2) + \ldots + 1] = (n - 1) + n(n+1)/2$,

- 

- Best case : O (n log n) -> Basically great pivots
- Worst case: O(n^2) -> Basically terrible pivots

- The constants have to do with the number of computer operations, so O(100n) > O(50n)
- Also looking at probability, bad inputs for quick sort are pretty rare
- Therefore quick sort is not as bad as it looks lol :)