LECTURE 18

COMPARISON SORTS

MCS 275 Spring 2022

Emily Dumas

LECTURE 18: COMPARISON SORTS

Course bulletins:

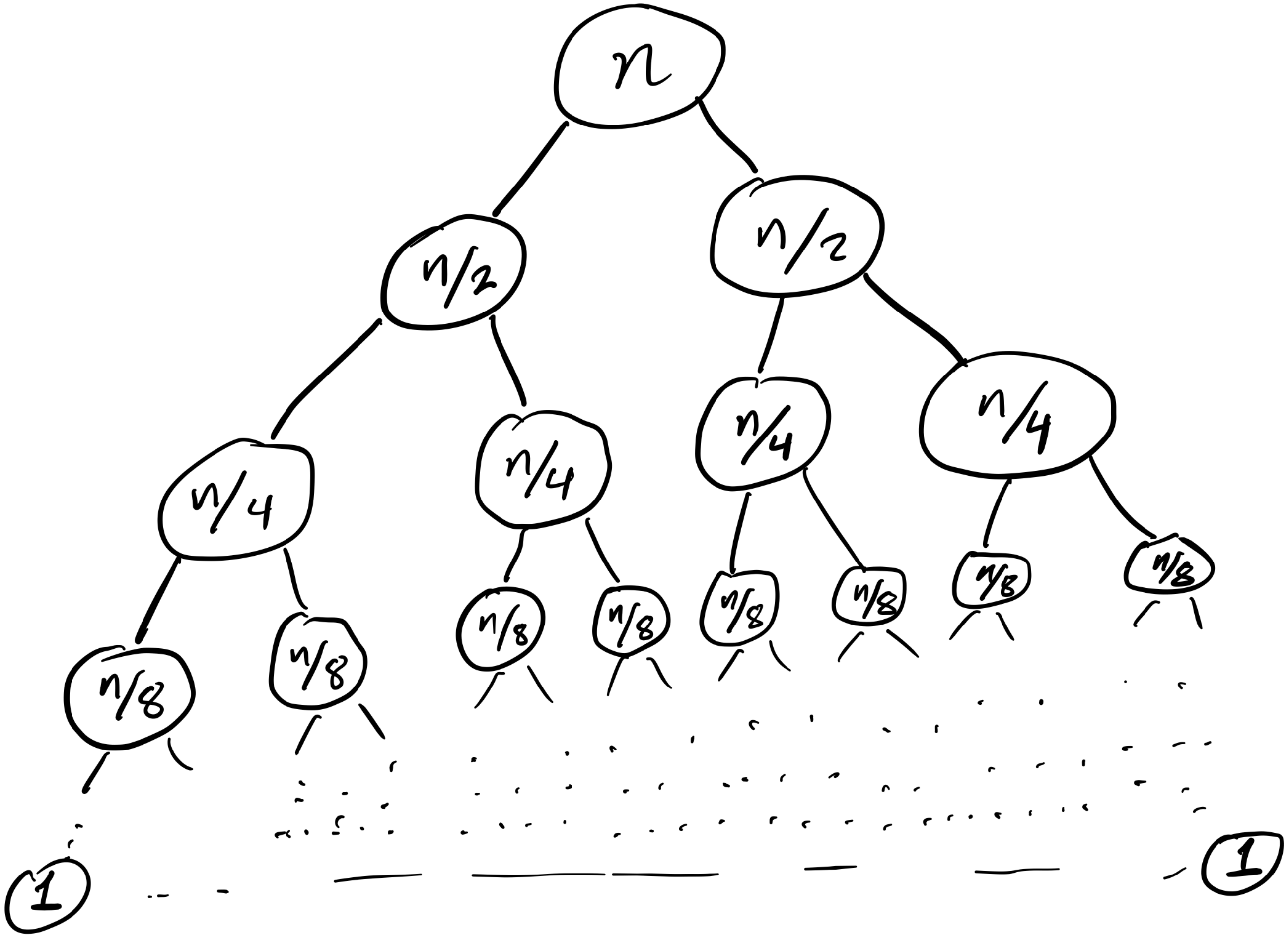
- Homework 6 due at Noon tomorrow.
- Project 2 due 6pm Friday. Autograder open.
- I cannot promise any availability between 1pm and 6pm on Friday. Plan accordingly.
- Worksheet 7 coming soon.
- Starting new topic (trees) on Wednesday.

GROWTH RATES

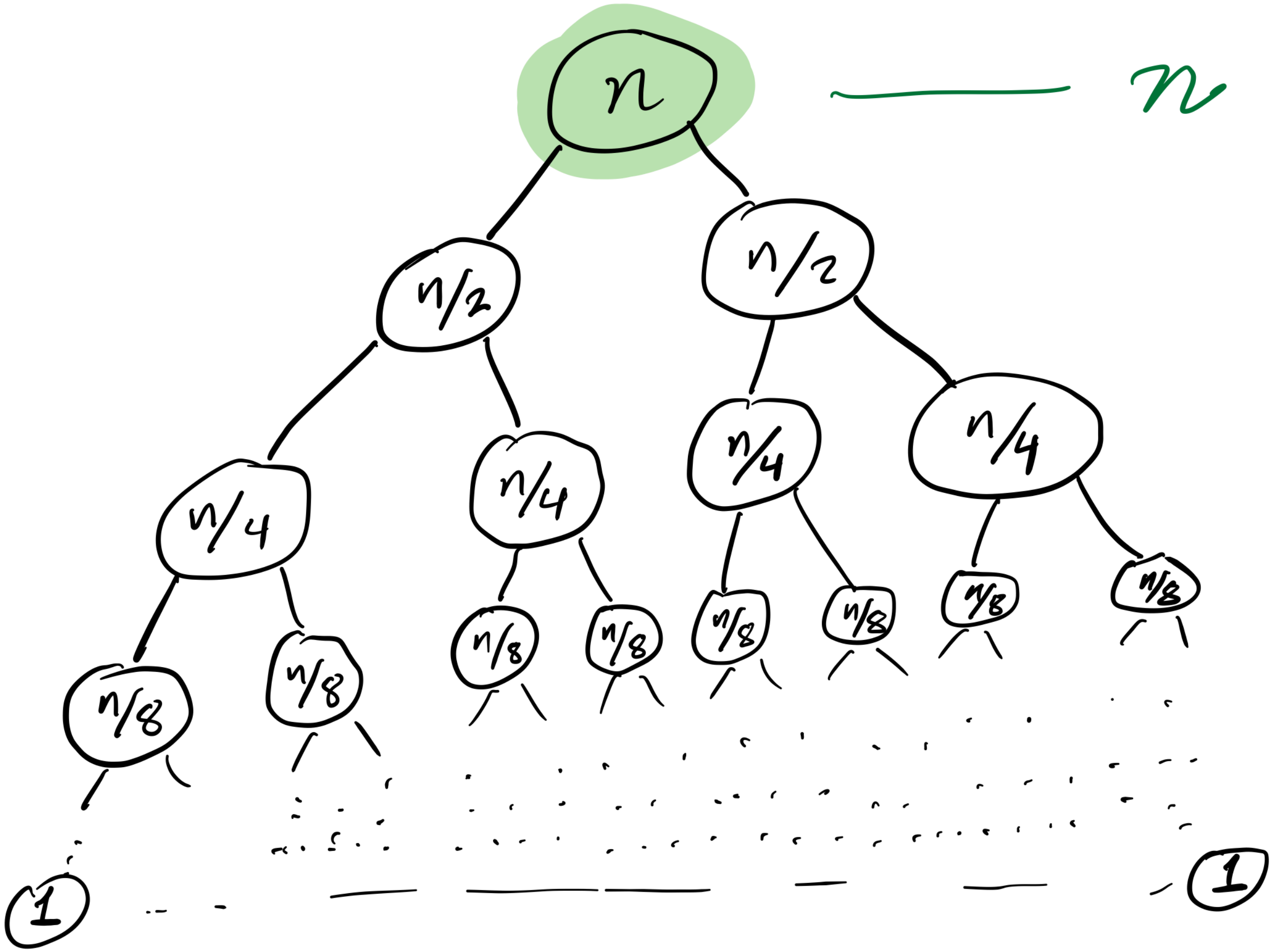
Let's look at the functions $n, n\log(n)$, and n^2 as n grows.

MERGESORT RECURSION TREE

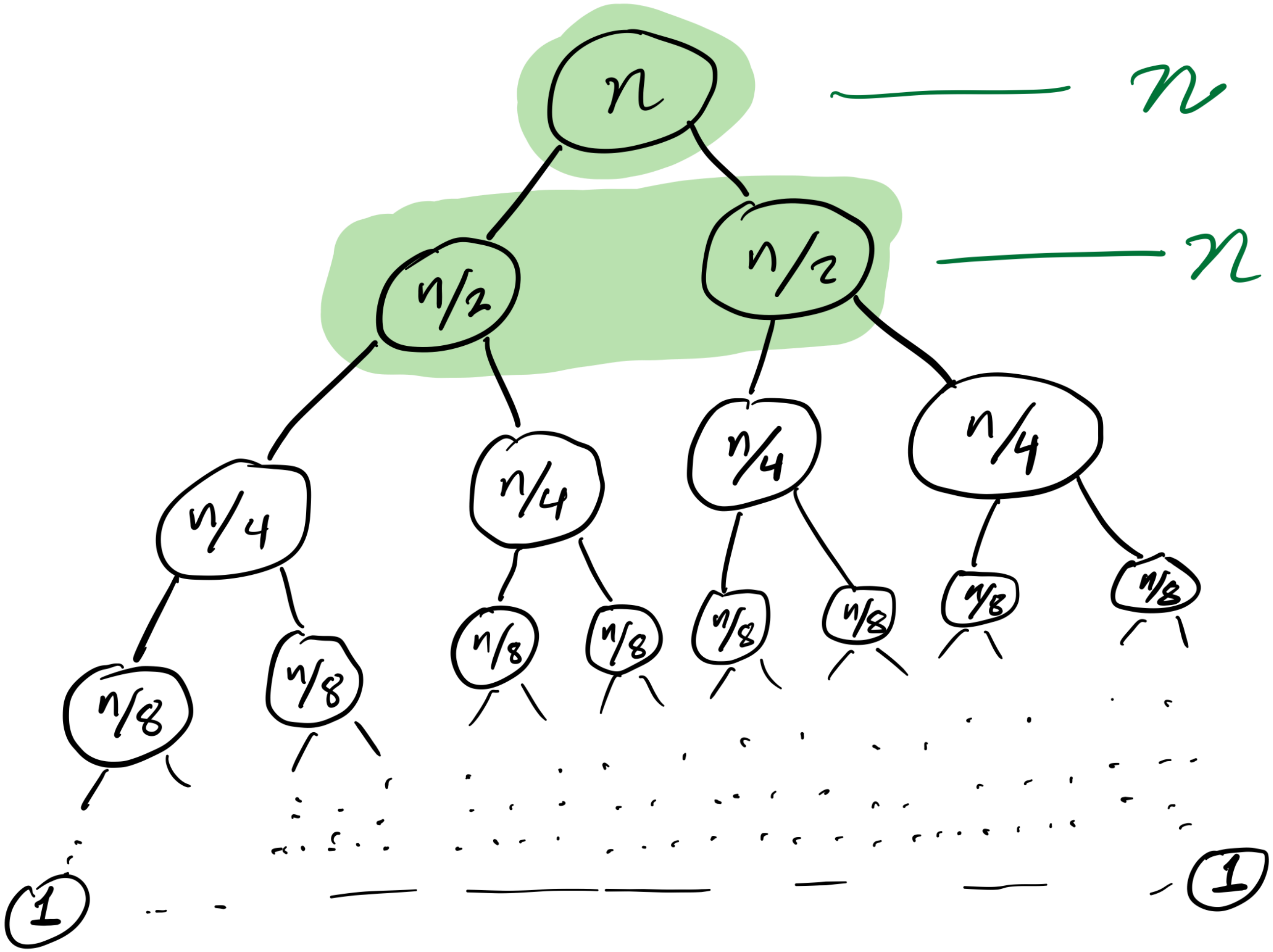
MERGESORT RECURSION TREE



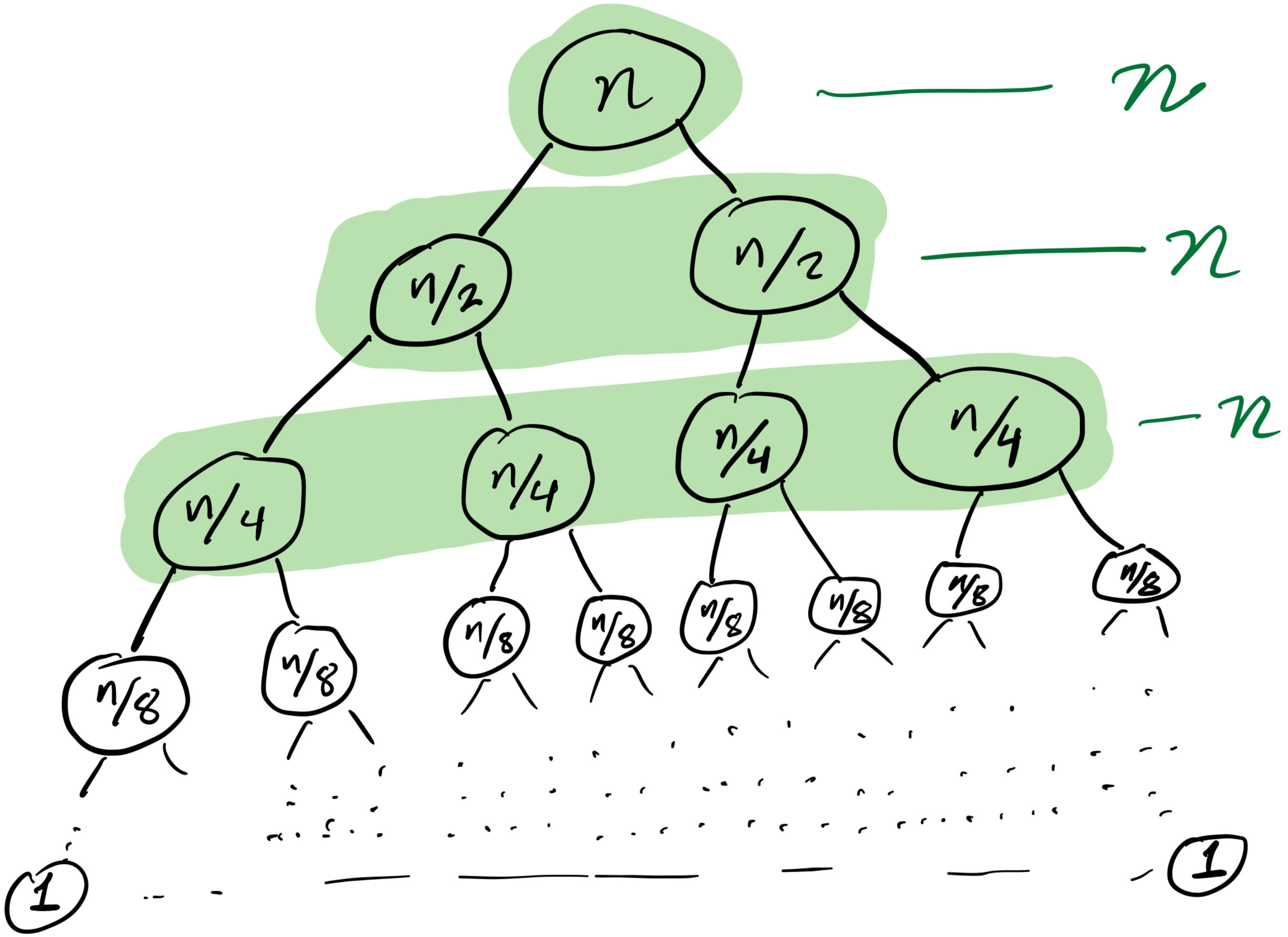
MERGESORT RECURSION TREE



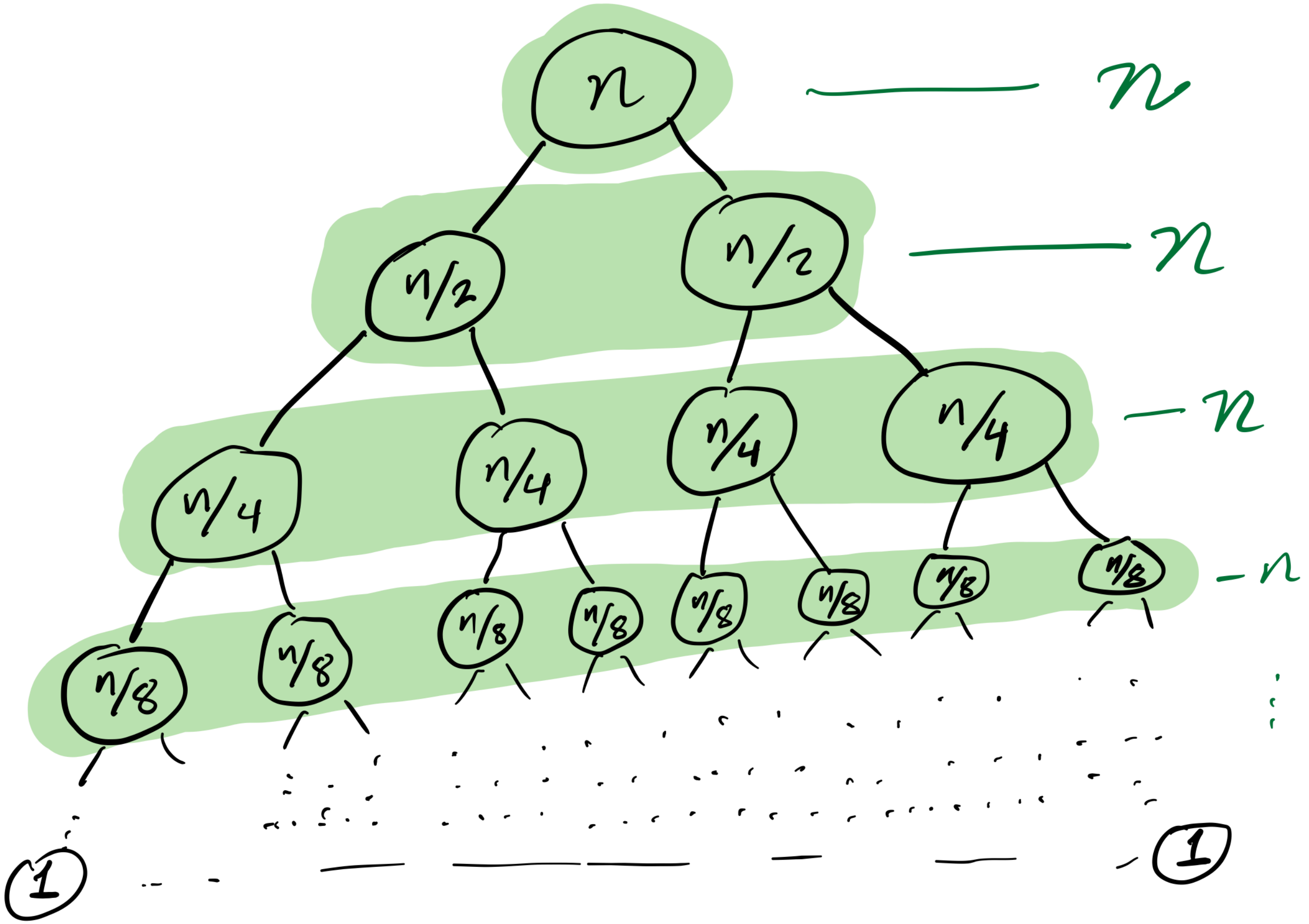
MERGESORT RECURSION TREE



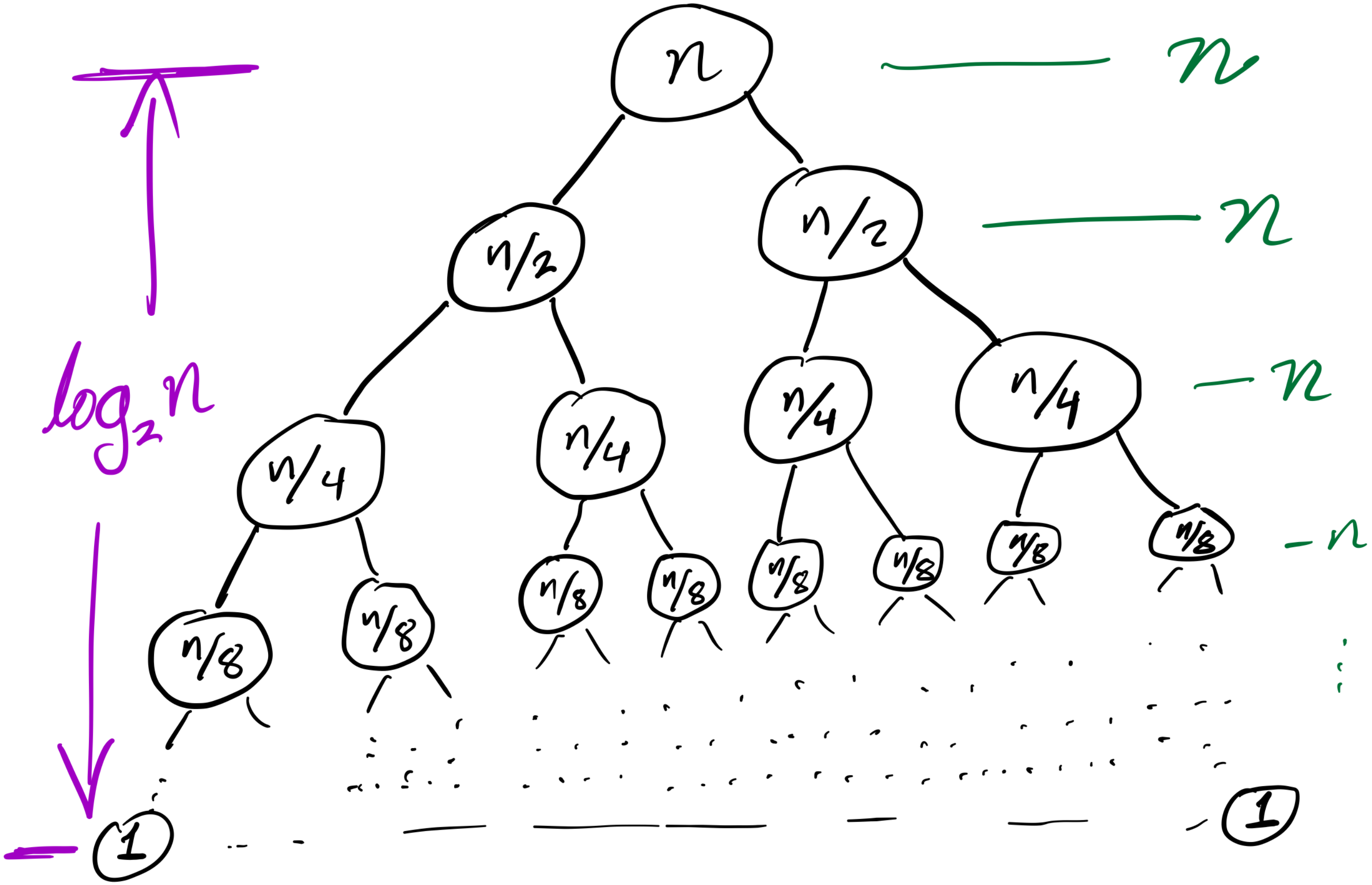
MERGESORT RECURSION TREE



MERGESORT RECURSION TREE



MERGESORT RECURSION TREE



MERGESORT RECURSION TREE

$$(\log_2 n)(n)$$

levels per level.

MERGESORT RECURSION TREE

$n \log_2 n$

MERGESORT RECURSION TREE

$C n \log n$

time cost of mergesort
for a list of length n.

EFFICIENCY

Theorem: If you measure the time cost of mergesort in any of these terms

- Number of comparisons made
- Number of assignments (e.g. `L[i] = x` counts as 1)
- Number of Python statements executed

then the cost to sort a list of length n is less than $Cn\log(n)$, for some constant C that only depends on which expense measure you chose.

LOOKING BACK ON QUICKSORT

It ought to be called **partition**sort because the algorithm is simply:

- Partition the list
- Quicksort the part before the pivot
- Quicksort the part after the pivot

OTHER PARTITION STRATEGORIES

We used the last element of the list as a pivot. Other popular choices:

- The first element, `L[start]`
- A random element of `L[start:end]`
- The element `L[(start+end)//2]`
- An element near the median of `L[start:end]` (more complicated to find!)

HOW TO CHOOSE?

Knowing something about your starting data may guide choice of partition strategy (or even the choice to use something other than quicksort).

Almost-sorted data is a common special case where first or last pivots are bad.

EFFICIENCY

Theorem: If you measure the time cost of quicksort in any of these terms

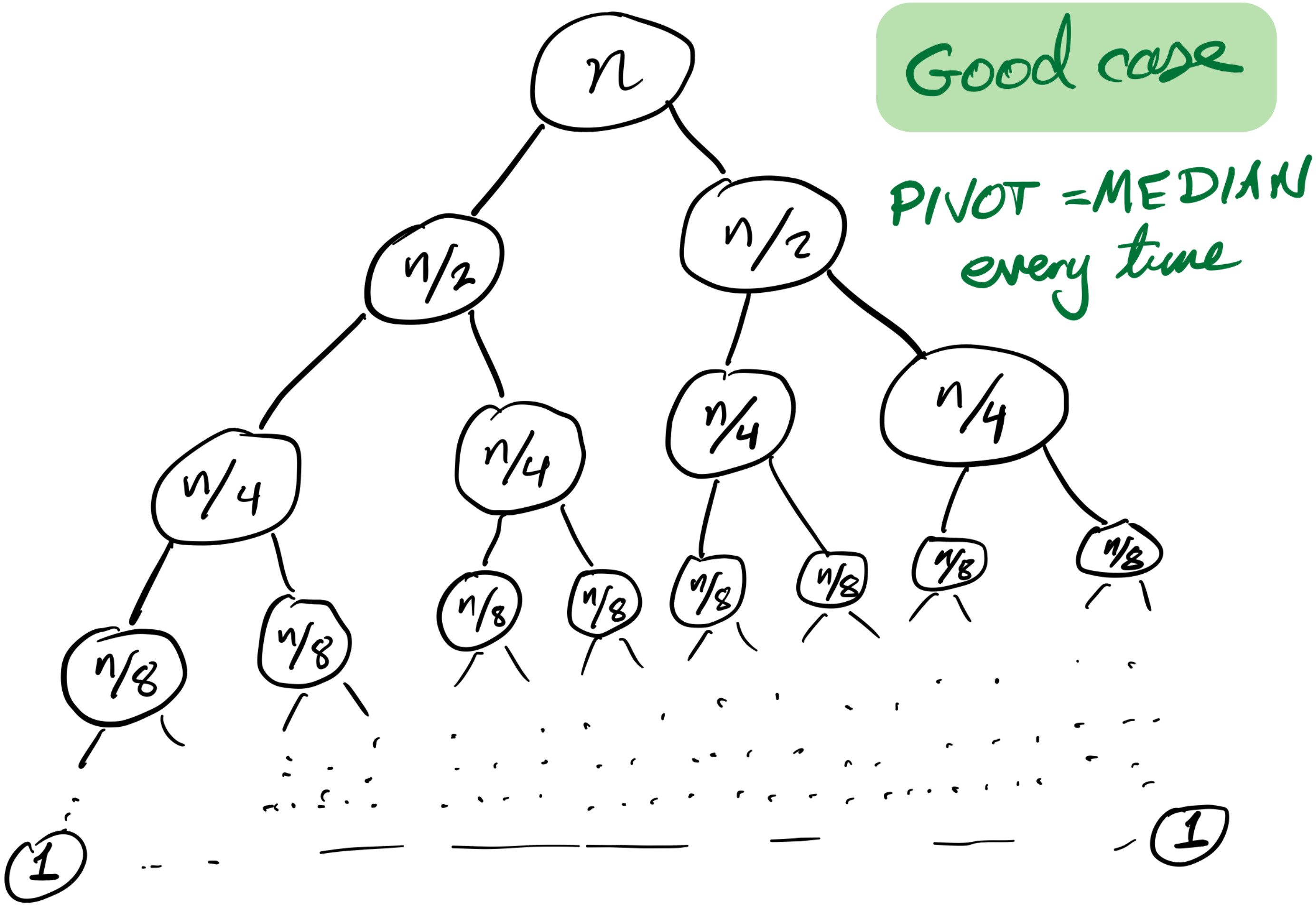
- Number of comparisons made
- Number of swaps or assignments
- Number of Python statements executed

then the cost to sort a list of length n is less than Cn^2, for some constant C.

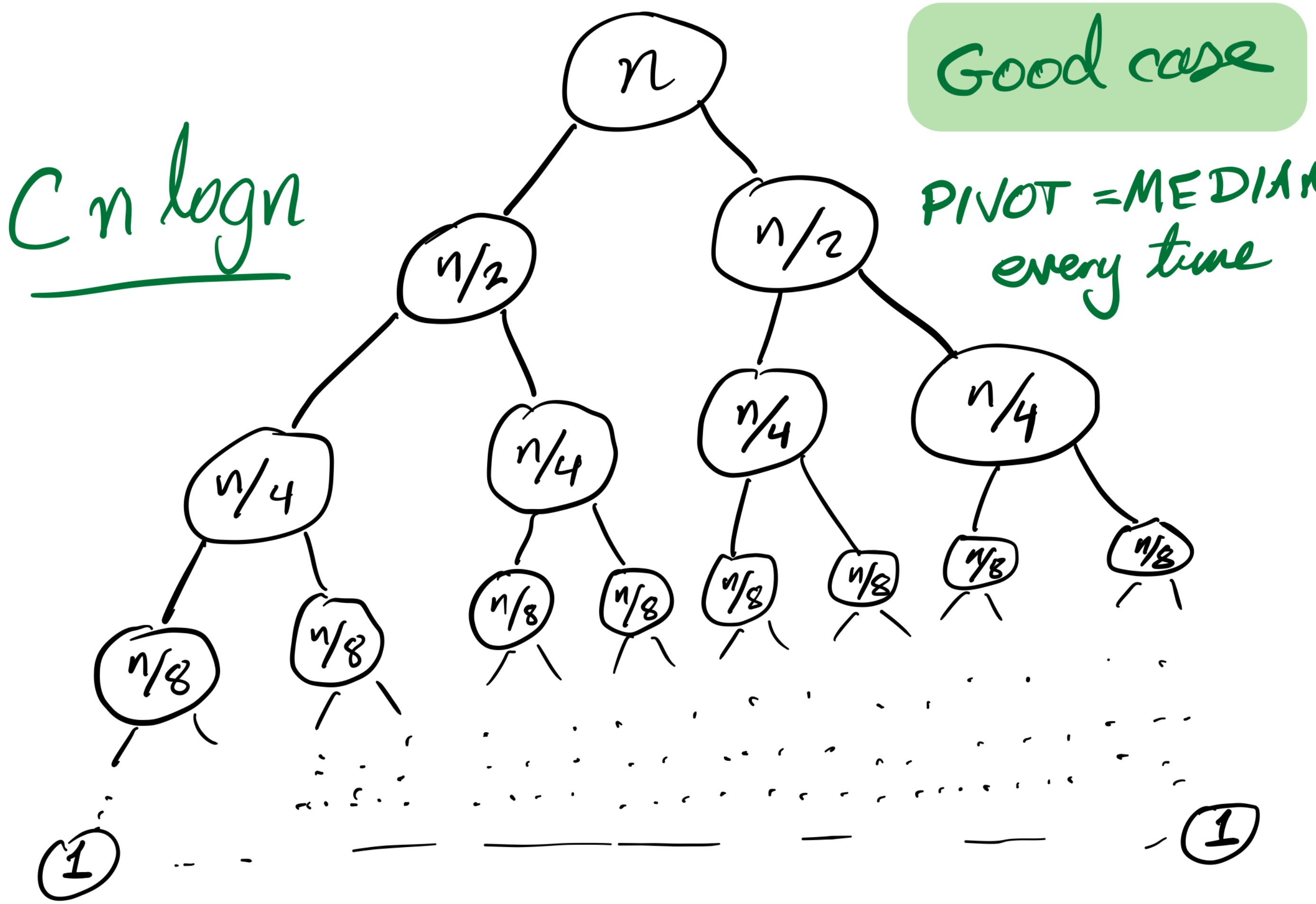
But if you average over all possible orders of the input data, the result is less than $Cn\log(n)$.

QUICKSORT RECURSION TREE

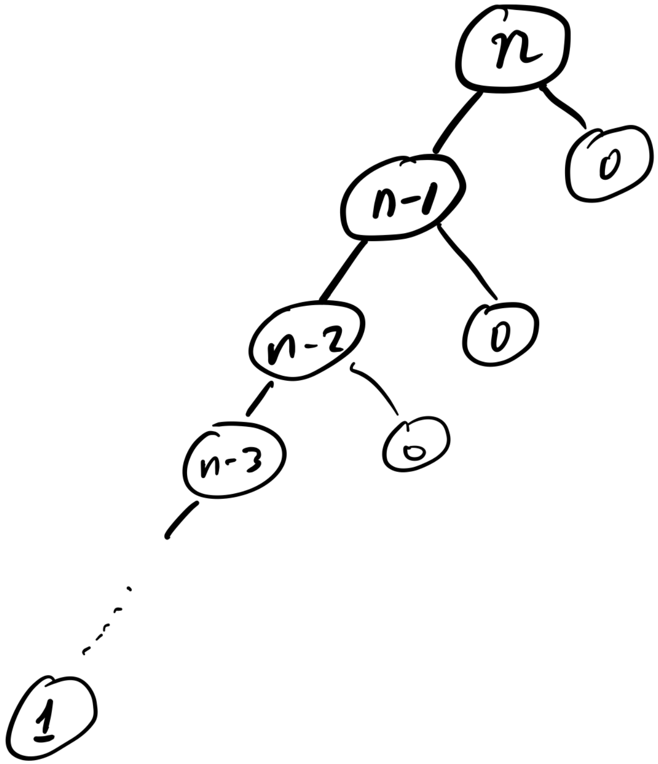
QUICKSORT RECURSION TREE



QUICKSORT RECURSION TREE



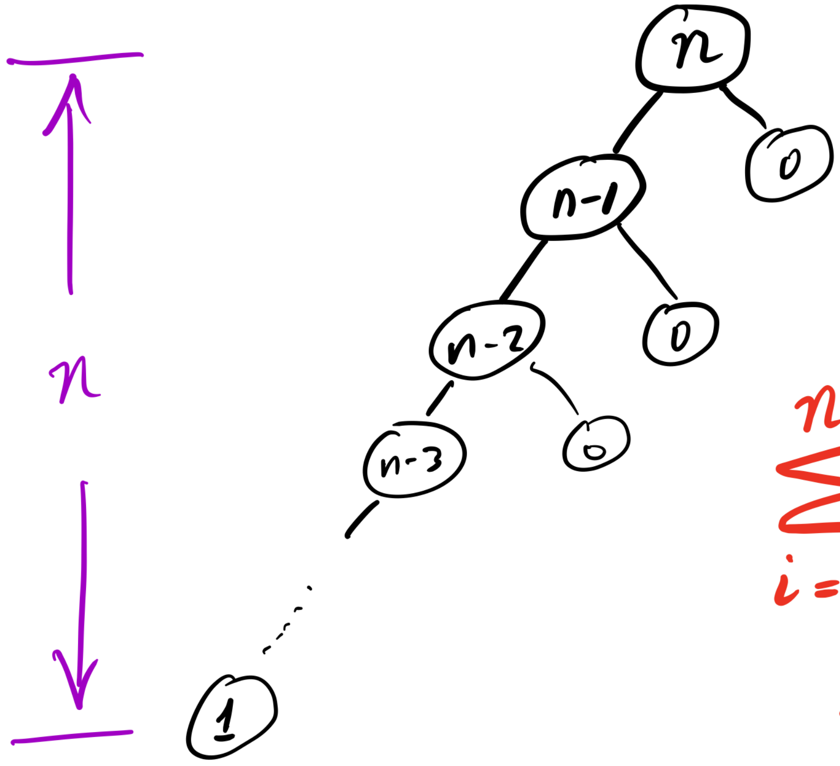
QUICKSORT RECURSION TREE



WORST CASE

Pivot = max or min every time.

QUICKSORT RECURSION TREE



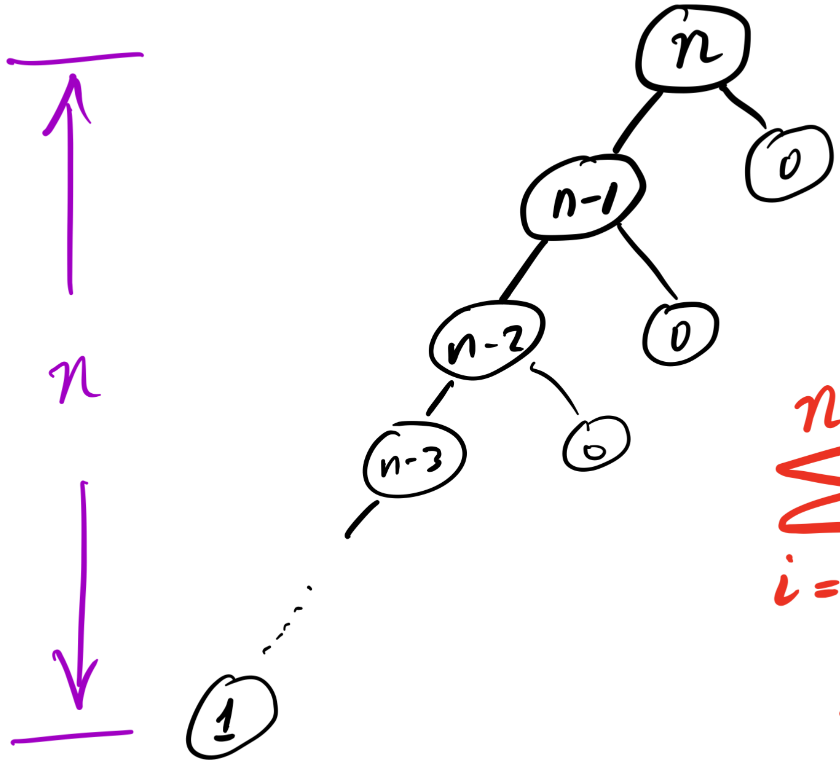
WORST CASE

Pivot = max or min every time.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$= Cn^2 + \text{smaller terms}$$

QUICKSORT RECURSION TREE



WORST CASE

Pivot = max or min every time.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$= Cn^2 + \text{smaller terms}$$

SORTED INPUT + LAST ELT PIVOT DOES THIS !!

BAD CASE

What if we ask our version of `quicksort` to sort a list that is already sorted?

Recursion depth is n (whereas if the pivot is always the median it would be $\approx \log_2 n$).

Number of comparisons $\approx Cn^2$. Very slow!

STABILITY

A sort is called **stable** if items that compare as equal stay in the same relative order after sorting.

This could be important if the items are more complex objects we want to sort by one attribute (e.g. sort alphabetized employee records by hiring year).

As we implemented them:

- Mergesort is stable
- Quicksort is not stable

EFFICIENCY SUMMARY

Algorithm	Time (worst)	Time (average)	Stable?	Space
Mergesort	$Cn\log(n)$	$Cn\log(n)$	Yes	Cn
Quicksort	Cn^2	$Cn\log(n)$	No	C

(Every time C is used, it represents a different constant.)

OTHER COMPARISON SORTS

- **Insertion sort** — Convert the beginning of the list to a sorted list, starting with one element and growing by one element at a time.
- **Bubble sort** — Process the list from left to right. Any time two adjacent elements are in the wrong order, switch them. Repeat n times.

EFFICIENCY SUMMARY

Algorithm	Time (worst)	Time (average)	Stable?	Space
Mergesort	$Cn\log(n)$	$Cn\log(n)$	Yes	Cn
Quicksort	Cn^2	$Cn\log(n)$	No	C
Insertion	Cn^2	Cn^2	Yes	C
Bubble	Cn^2	Cn^2	Yes	C

(Every time C is used, it represents a different constant.)

CLOSING THOUGHTS ON SORTING

Mergesort is rarely a bad choice. It is stable and sorts in $Cn\log(n)$ time. Nearly sorted input is not a pathological case. Its main weakness is its use of memory proportional to the input size.

Heapsort, which we may discuss later, has $Cn\log(n)$ running time and uses constant space, but it is not stable.

There are stable comparison sorts with $Cn\log(n)$ running time and constant space (best in every category!) though they tend to be more complex.

If swaps and comparisons have very different cost, it may be important to select an algorithm that minimizes one of them. Python's `list.sort` assumes that comparisons are expensive, and uses Timsort.

QUADRATIC DANGER

Algorithms that take time proportional to n^2 are a big source of real-world trouble. They are often fast enough in small-scale tests to not be noticed as a problem, yet are slow enough for large inputs to disable the fastest computers.

REFERENCES

Unchanged from Lecture 17

- Making nice visualizations of sorting algorithms is a cottage industry in CS education. Some you might like to check out:
  - 2D visualization through color sorting by Linus Lee
  - Animated bar graph visualization of many sorting algorithms by Alex Macy
  - Slanted line animated visualizations of mergesort and quicksort by Mike Bostock

REVISION HISTORY

- 2022-02-21 Initial publication