Analysis of Quick Sort

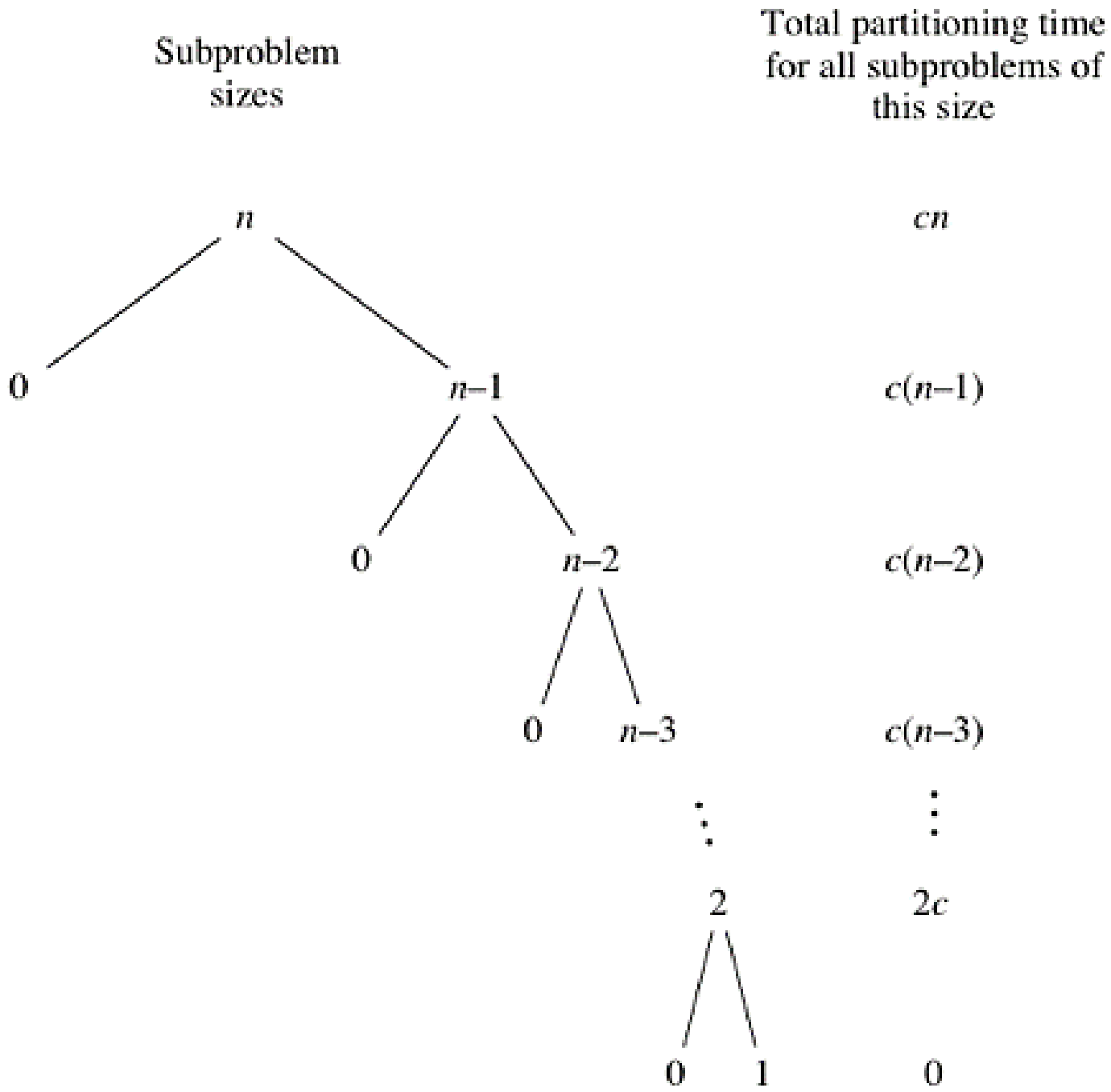
(Class 14)

From Book's Chapter 7

- We studied the worst-case running time of quick sort in previous class.
- The worst-case will occur when 2 conditions are true:
  - The input array is already sorted.
  - We pick the 1st element as pivot at each recursion.

Worst Case Running Time Recursion Tree

- When quicksort always has the most unbalanced partitions possible.
- Then the original call takes cn time for some constant c.
- The recursive call on $n - 1$ elements take $c(n - 1)$ time.
- The recursive call on $n - 2$ elements take $c(n - 2)$ time, and so on.
- Here's a tree of the subproblem sizes with their partitioning times:


Subproblem sizes
Total partitioning time for all subproblems of this size
n
cn
0
n–1
c(n–1)
0
n–2
c(n–2)
0
n–3
c(n–3)
2
2c
0
1
0

- When we total up the partitioning times for each level, we get:

$$cn + c(n-1) + c(n-2) + \cdots + 2c$$

$$= c(n + (n-1) + (n-2) + \cdots + 2)$$

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

$$\boldsymbol{T(n) = O(n^2)}$$

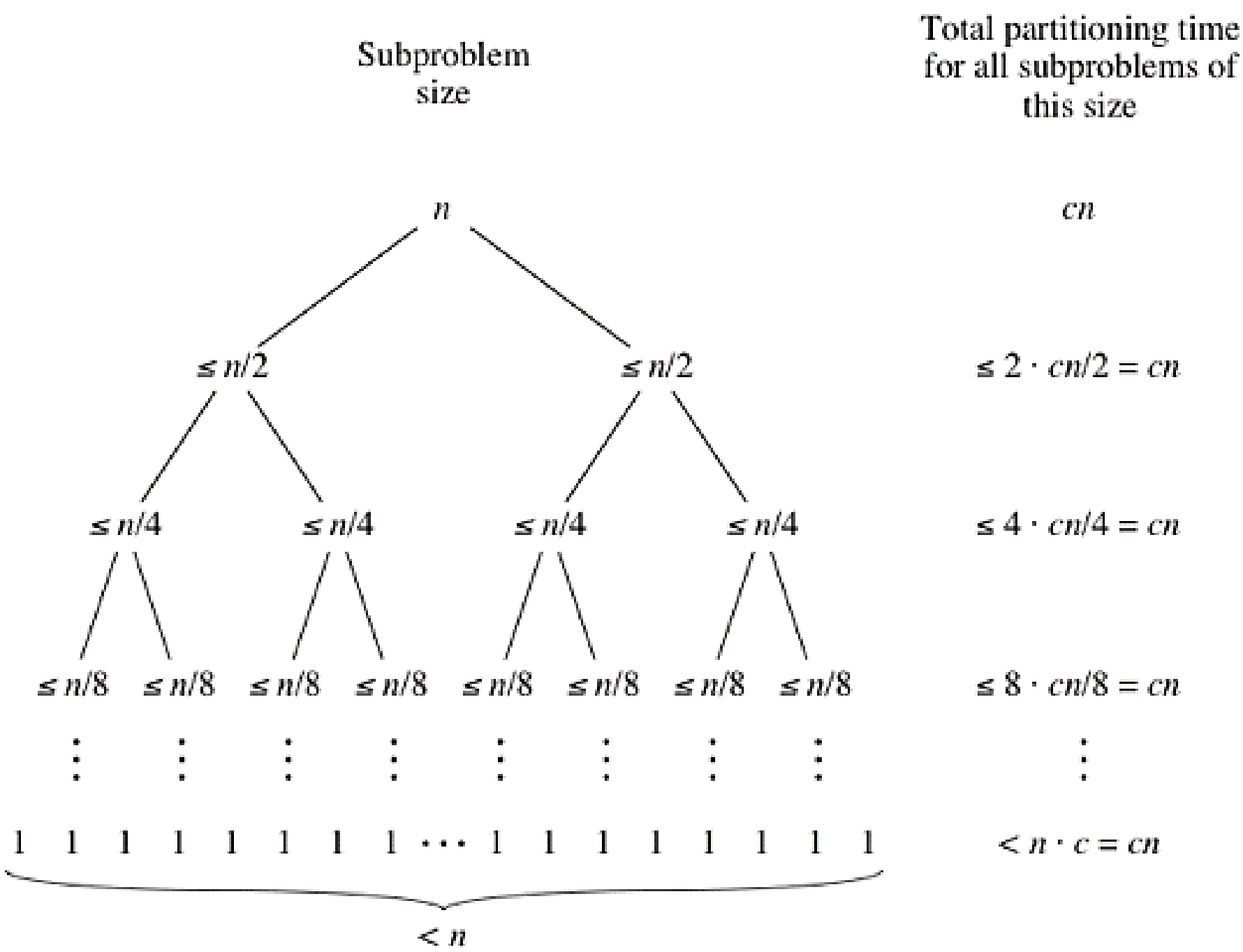
- The last line is because $1 + 2 + 3 + \cdots + n$ is the arithmetic series.
- We ignore the constant term c.
- In big-O notation, quicksort's worst-case running time is $O(n^2)$.

Best Case Running Time Recursion Tree

- Quicksort's best case occurs when the partitions are as evenly balanced as possible.
- Their sizes either are equal or differ by 1 element.
- The 1st case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has $\frac{n-1}{2}$ elements.
- The 2nd case occurs if the subarray has an even number of elements, and one partition has $\frac{n}{2}$ elements with the other having $(\frac{n}{2} - 1)$.

- In either of these cases, each partition has at most $\frac{n}{2}$ elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times.
- Using big-O notation, we get the same result as for merge sort.

$$T(n) = O(n \log n)$$


Subproblem size
Total partitioning time for all subproblems of this size
n
cn
≤ n/2
≤ n/2
≤ 2 · cn/2 = cn
≤ n/4
≤ n/4
≤ n/4
≤ n/4
≤ 4 · cn/4 = cn
≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8 ≤ n/8
≤ 8 · cn/8 = cn
1 1 1 1 1 1 1 1 ··· 1 1 1 1 1 1 1 1
< n · c = cn
< n

Average-Case Analysis of Quicksort

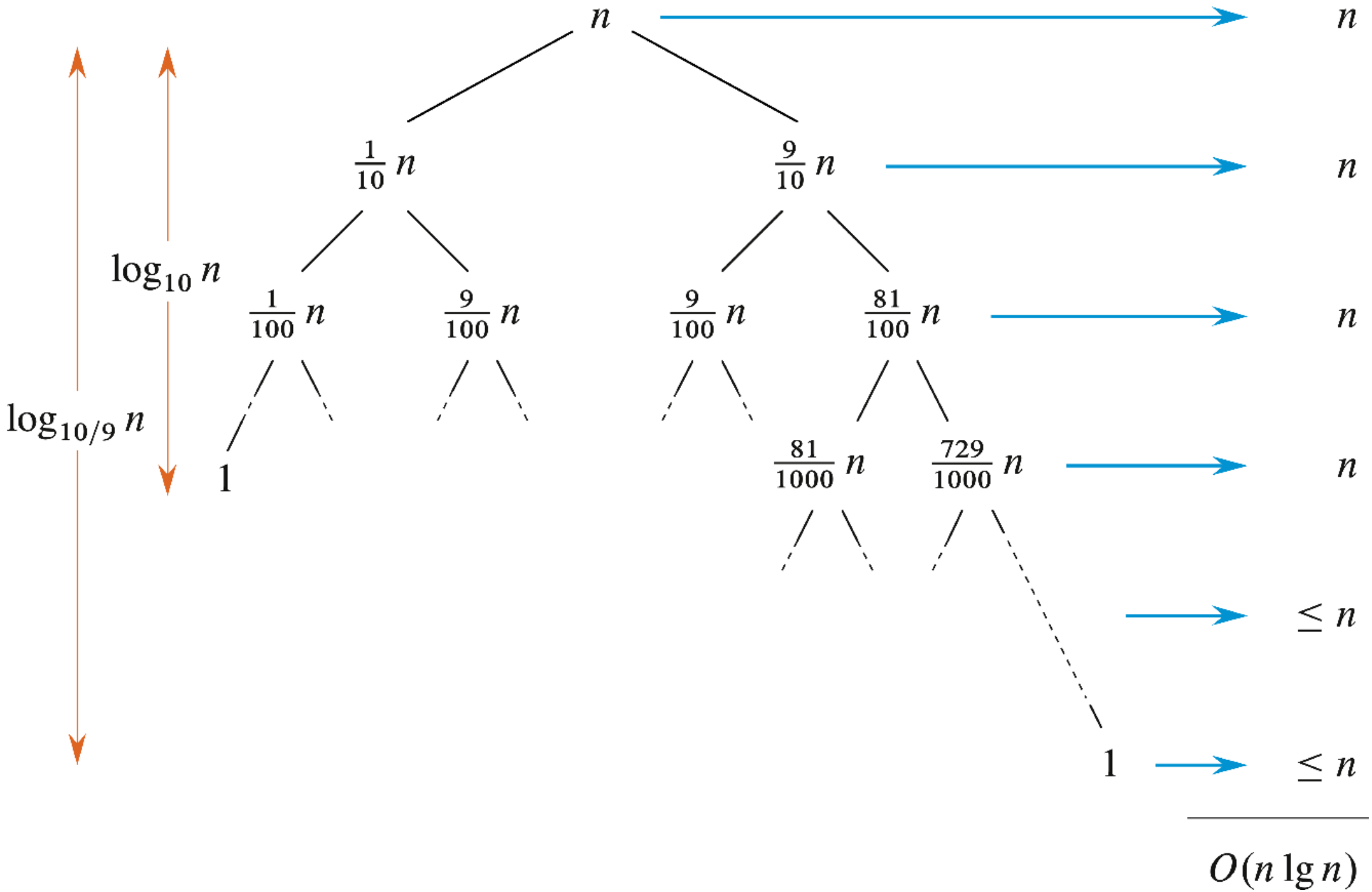
- We will now show that in the average case, quicksort runs also in $O(n \log n)$ time.
- Recall that when we talked about average case at the beginning of the semester.
- We said that it depends on some assumption about the distribution of inputs.
- It depends upon the probability.

- However, in the case of quicksort, the analysis does not depend on the distribution of input at all.
- It only depends upon the random choices of pivots that the algorithm makes.
- The algorithm has n random choices for the pivot element.
- Each of these choices has an equal probability $\frac{1}{n}$ of occurring.

Average Case Running Time Recursion Tree

- Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at seems quite unbalanced.
- We then obtain the recurrence:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$


n
n
1/10 n
9/10 n
n
log10 n
1/100 n
9/100 n
9/100 n
81/100 n
n
log10/9 n
81/1000 n
729/1000 n
n
1
≤ n
1
≤ n
O(n lg n)

- Every level of the tree has cost n, until the recursion bottoms out in a base case at depth $log_{10}n$. And then the levels have cost at most n.
- The recursion terminates at depth $log_{\frac{10}{9}}n$.
- Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems highly unbalanced, quicksort runs in $O(n \log n)$ times asymptotically the same as if the split were right down the middle.

- Indeed, even a 99-to-1 split yields an $O(n \log n)$ running time.
- In fact, any split of *constant* proportionality yields a recursion tree of depth $\log n$, where the cost at each level is n.
- The running time is therefore $O(n \log n)$ whenever the split has constant proportionality.

- The left child of each node represents a subproblem size $\frac{1}{10}$ as large, and the right child represents a subproblem size $\frac{9}{10}$ as large.
- Since the smaller subproblems are on the left, by following a path of left children, we get from the root down to a subproblem size of 1 faster than along any other path.
- As the figure shows, after $log_{10}n$ levels, we get down to a subproblem size of 1 in left subarray.

Why $\boldsymbol{log_{10}n}$ Levels on Left Subtree?

- It might be easiest to think in terms of starting with a subproblem size of 1 and multiplying it by 10 until we reach n.
- In other words, we're asking for what value of x is $10^x = n$?
- The answer is $x = log_{10}n$.

Why $\boldsymbol{log_{\frac{10}{9}} n}$ Levels on Right Subtree?

- How about going down a path of right children?
- The figure shows that it takes $log_{\frac{10}{9}} n$ levels to get down to a subproblem of size 1.
- Why $log_{\frac{10}{9}} n$ levels?

- Since each right child is $\frac{9}{10}$ of the size of the node above it (its parent node), each parent is $\frac{10}{9}$ times the size of its right child.
- Let's again think of starting with a subproblem of size 1 and multiplying the size by $\frac{10}{9}$ until we reach n.
- For what value of x is $\left(\frac{10}{9}\right)^x = n$?
- The answer is $x = log_{\frac{10}{9}} n$.

- There are at most $log_{\frac{10}{9}}n$ levels and each level has running time as cn.
- So total running time is:

$$T(n) = cn(log_{\frac{10}{9}}n)$$

- Converting the log of base $10/9$ to log of base 2:

$$\log_a n = \frac{\log_b n}{\log_b a}$$

- for all positive numbers a, b and n. Letting $a = \frac{10}{9}$ and $b = 2$, we get:

$$\log_{10/9} n = \frac{\log_2 n}{\log_2 (10/9)}$$

$$\log_{10/9} n = \frac{\log_2 n}{0.152}$$

$$\log_{10/9} n \approx \log_2 n$$

$$T(n) \approx cn(log_2 n)$$

$$\boldsymbol{T(n) = O(n\, log_2 n)}$$

- So, $\log_{10/9}$ and $\log_2 n$ differ by only a factor of $\log_2(10/9) = 0.152$ which is a constant.
- Since constant factors don't matter when we use big-O notation.
- So, we can say that if all the splits are 9-to-1, then quicksort's running time is $O(n \log_2 n)$.

Randomized Quicksort

- Suppose that your worst enemy has given you an array to sort with quicksort, knowing that you always choose the leftmost element in each subarray as the pivot.
- And has arranged the array so that you always get the worst-case split.
- How can you save yourself in that condition?

- We could not necessarily choose the leftmost element in each subarray as the pivot.
- Instead, we could randomly choose an element in the subarray, and use that element as the pivot.
- But because the partition function assumes that the pivot is in the leftmost position of the subarray, so we just swap the element that we chose as the pivot with the leftmost element, and then partition as before.
- Unless your enemy knows how you choose random locations in the subarray, you win!

Algorithmic Animations

- We can view the process of the sorting algorithms using the animations.
- **Link:** https://visualgo.net/en/sorting
- **Link:** https://www.toptal.com/developers/sorting-algorithms