Lecture 6

Quick Sort

Md. Asif Bin Khaled

Quick Sort

The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of n numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $\Theta(n \lg n)$, and the constant factors hidden in the $\Theta(n \lg n)$ notation are quite small. It also has the advantage of sorting in place and it works well even in virtual-memory environments.

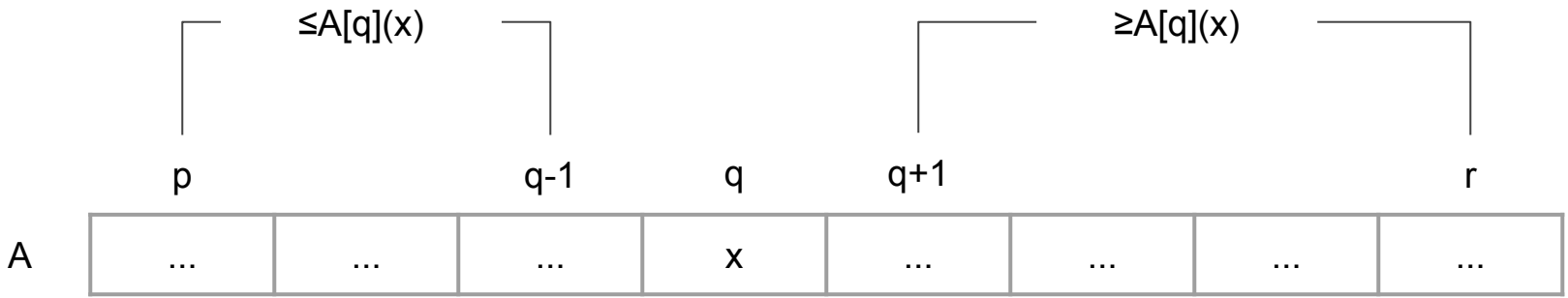
Quick Sort Continued

Quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray A[p...r]

1. **Divide**: Partition (rearrange) the array A[p...r] into two (possibly empty) subarrays A[p...q-1] and A[q+1...r] such that each element of A[p...q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1...r]. Compute the index q as part of this partitioning procedure.

Quick Sort Continued

2. **Conquer**: Sort the two subarrays A[p...q-1] and A[q+1...r] by recursive calls to quick sort.
3. **Combine**: Because the subarrays are already sorted, no work is needed to combine them, the entire array A[p...r] is now sorted



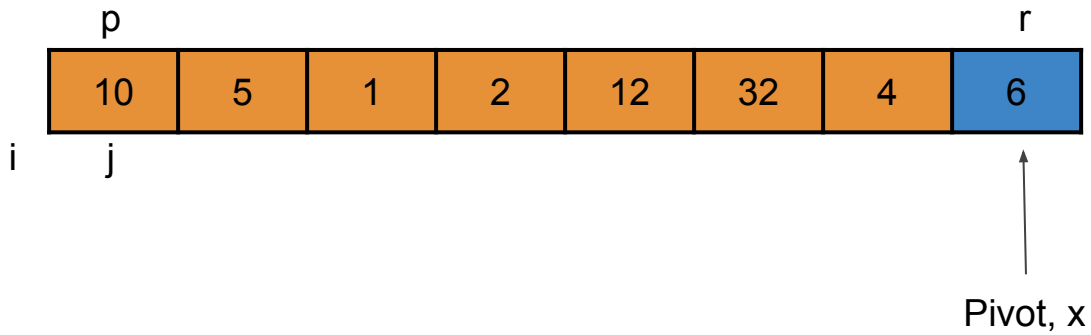
Partition Pseudocode

– – –

Partition(A,p,r):

```
    x=A[r]
    i=p-1
    for j=p to r-1:
        if A[j]≤x:
            i=i+1
            exchange A[i] with A[j]
    exchange A[i+1] with A[r]
    return i+1
```

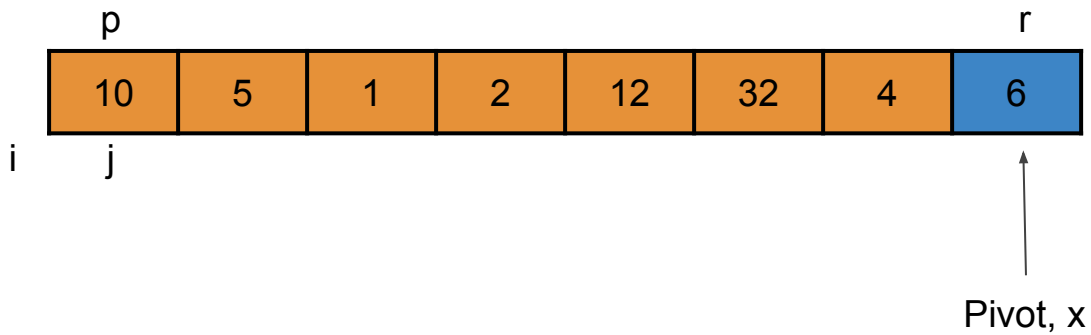
Simulation of Partition



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

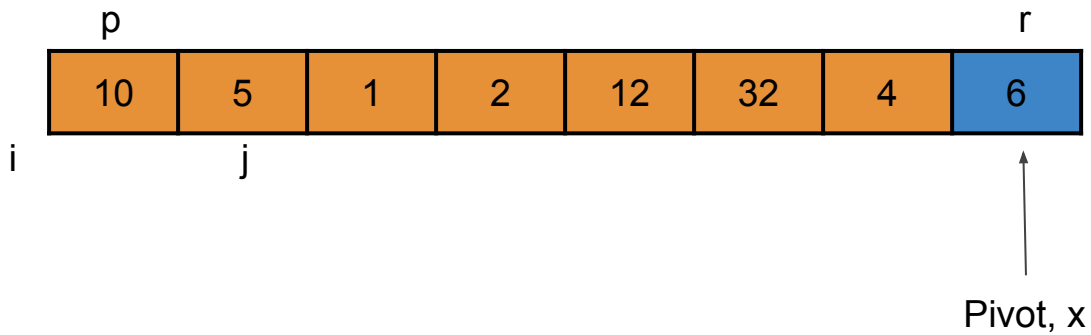
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

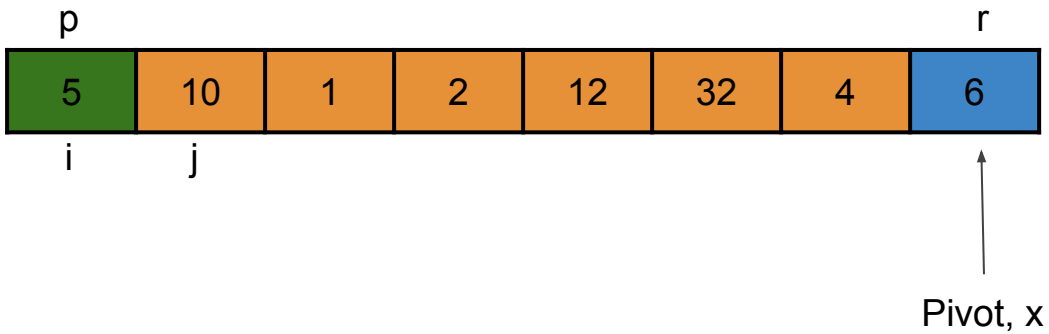
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

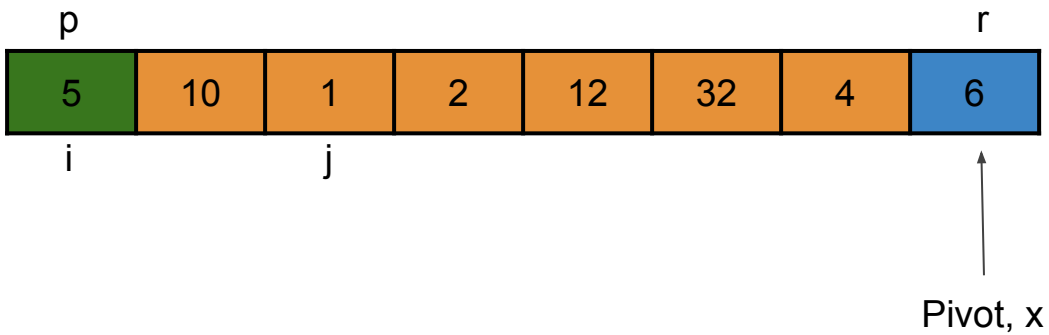
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

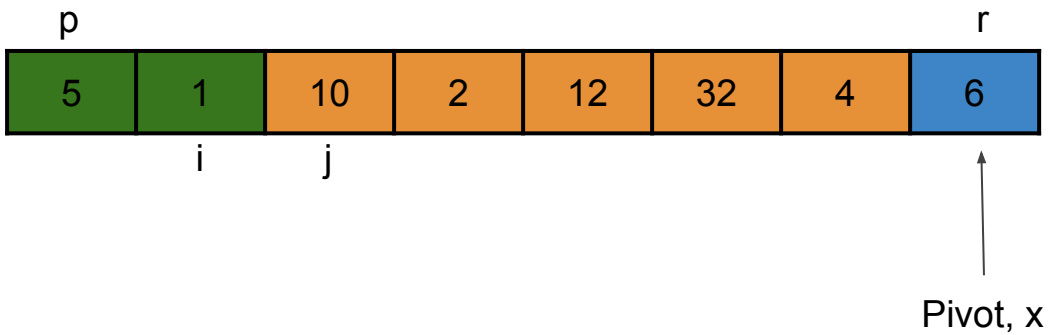
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

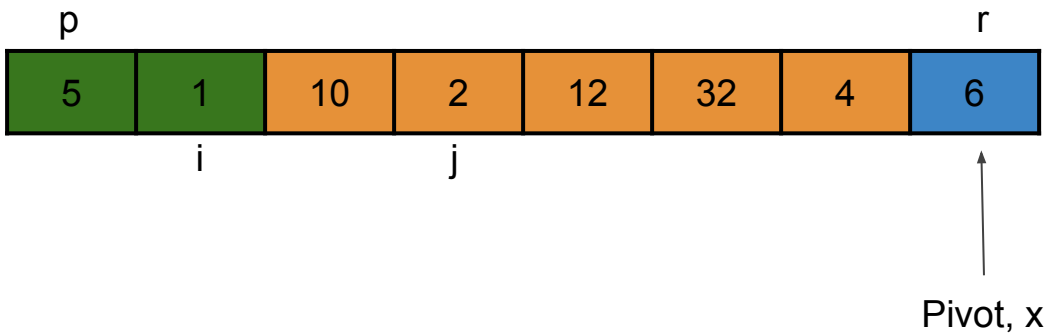
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

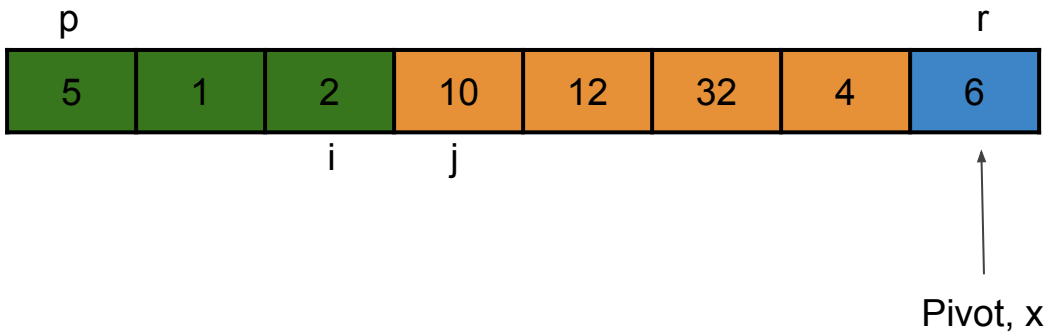
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

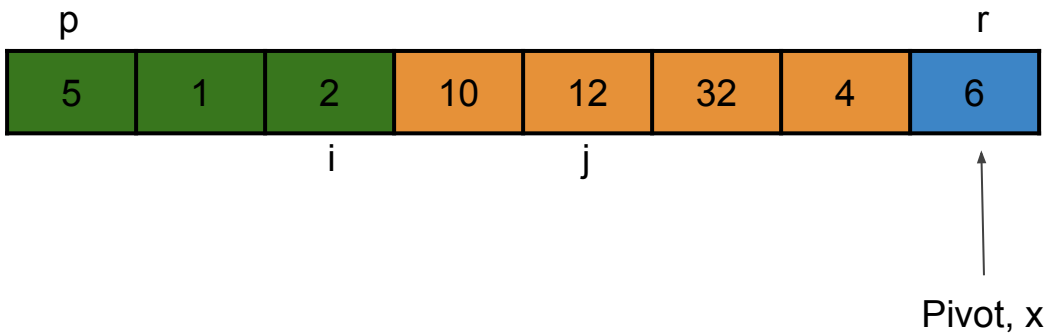
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

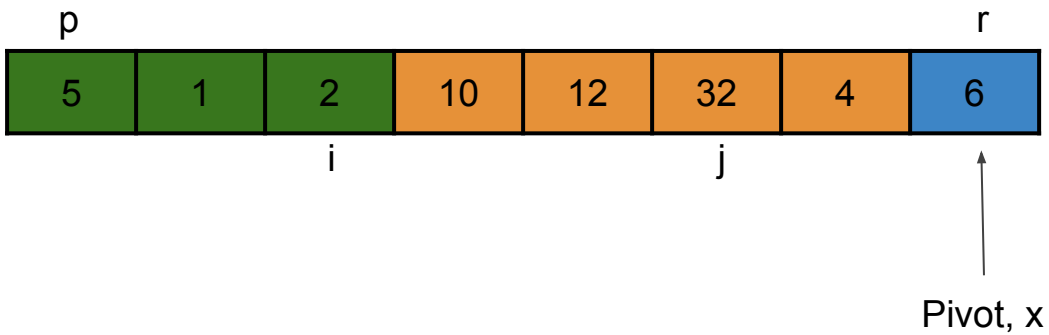
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

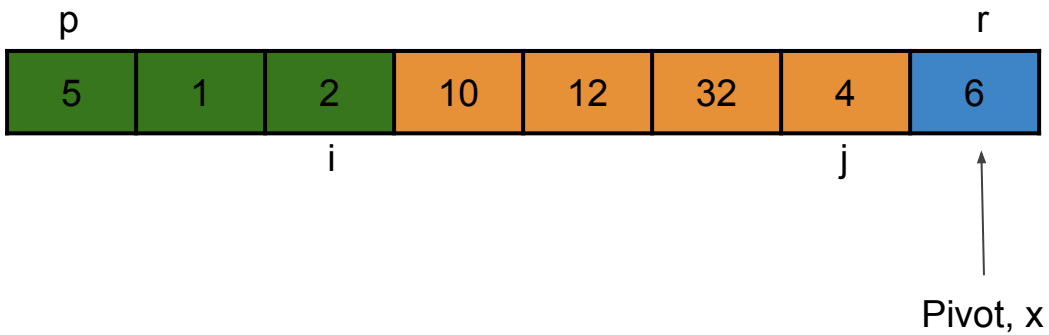
A[j] is less than or equal to x



Simulation of Partition

Comparing A[j] with x and swapping A[i] with

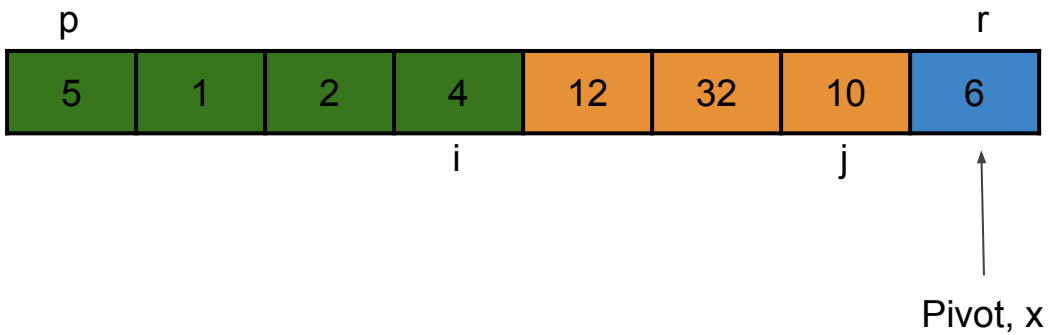
A[j] is less than or equal to x



Simulation of Partition

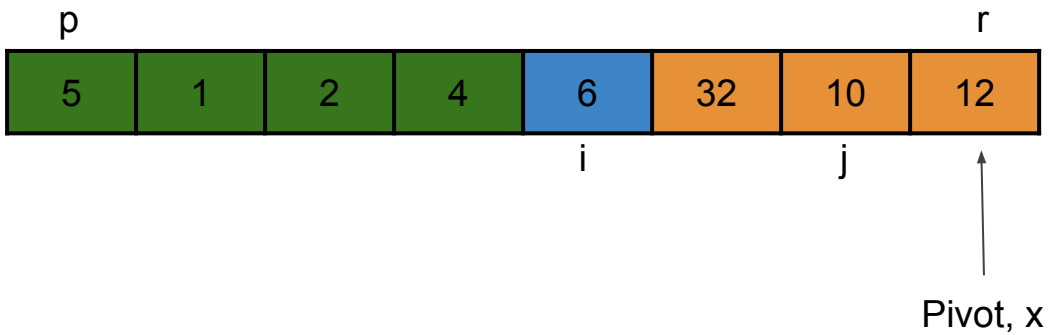
Comparing A[j] with x and swapping A[i] with

A[j] is less than or equal to x



Simulation of Partition

As we have compared all the elements till r-1 so now we exchange A[i+1] with A[r]



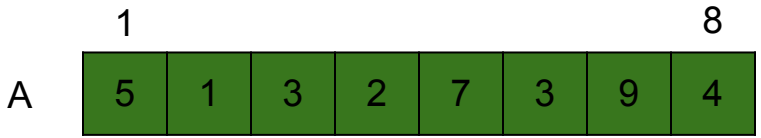
Quick Sort Pseudocode

– – –

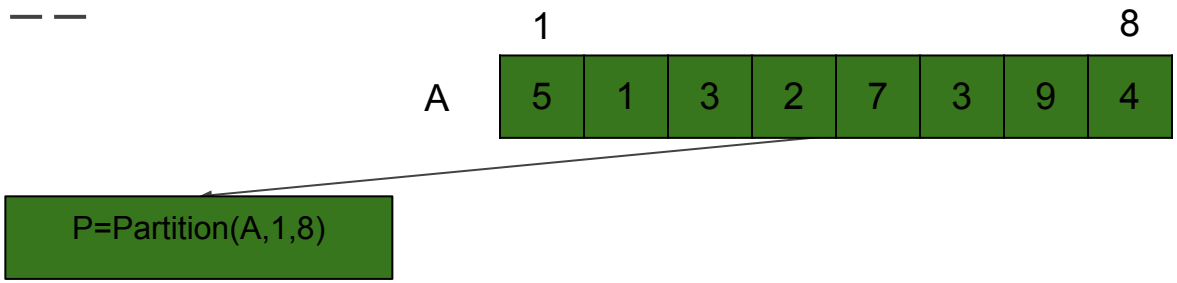
```
Quicksort(A,p,r):
    if p<r:
        q=Partition(A,p,r)
        Quicksort(A,p,q-1)
        Quicksort(A,q+1,r)
```

Simulation of Quick Sort

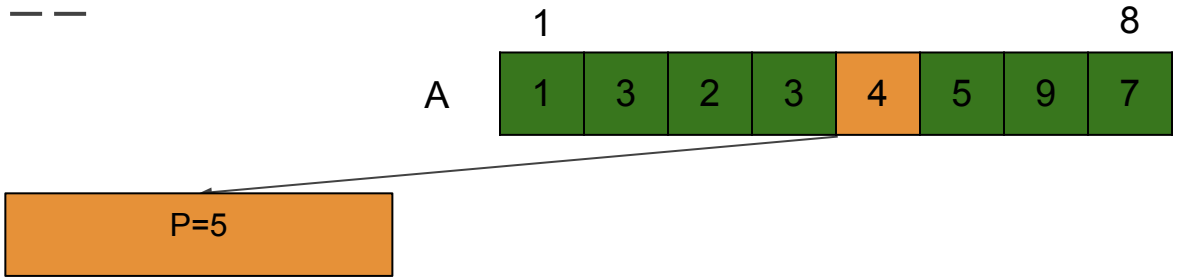
– – –



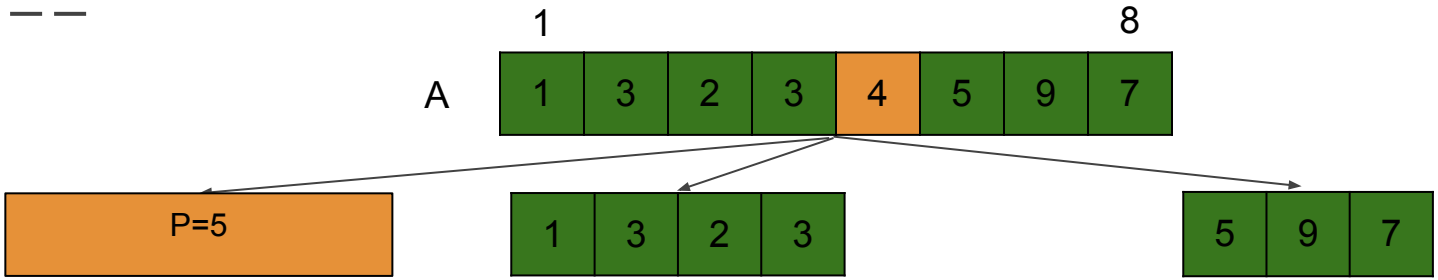
Simulation of Quick Sort



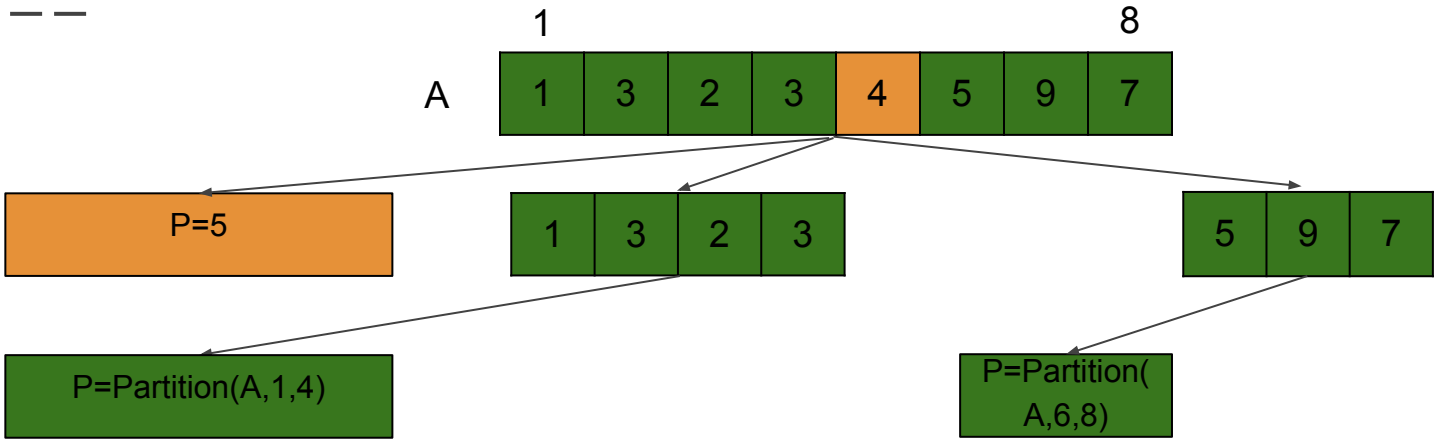
Simulation of Quick Sort



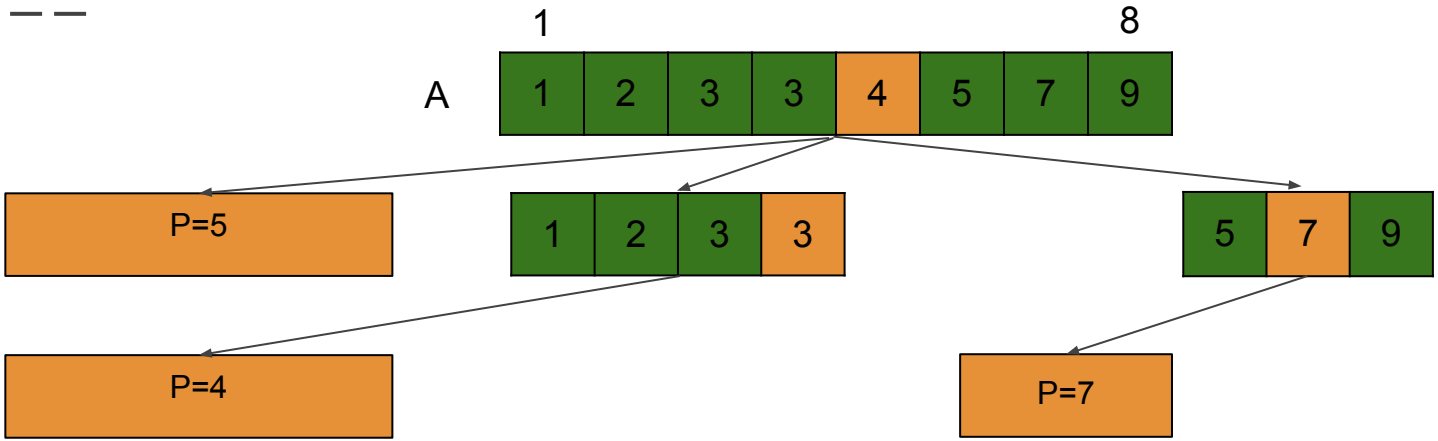
Simulation of Quick Sort



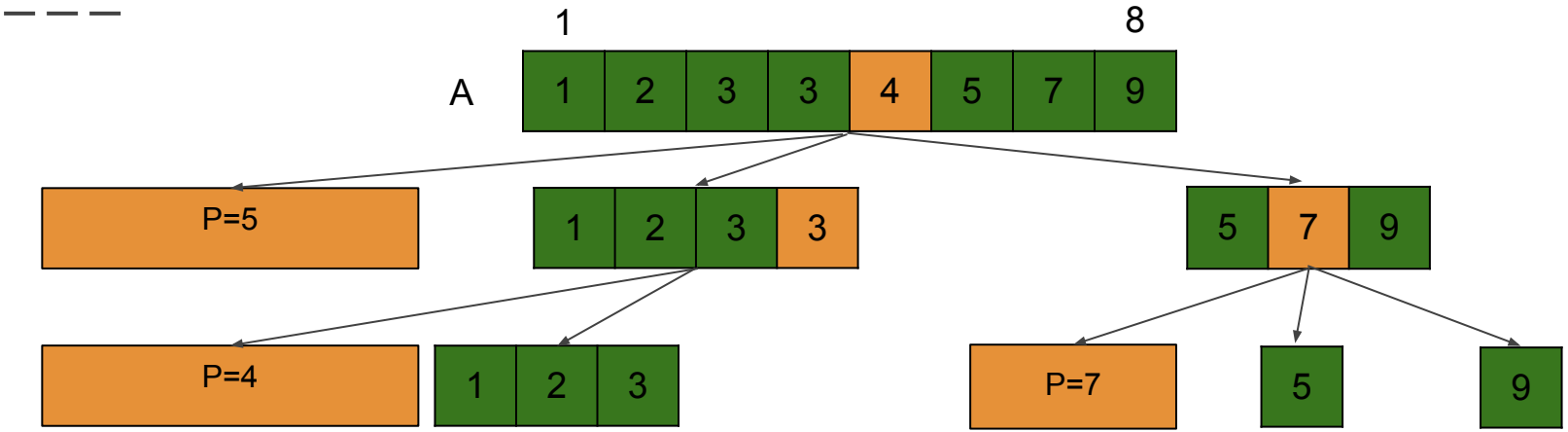
Simulation of Quick Sort



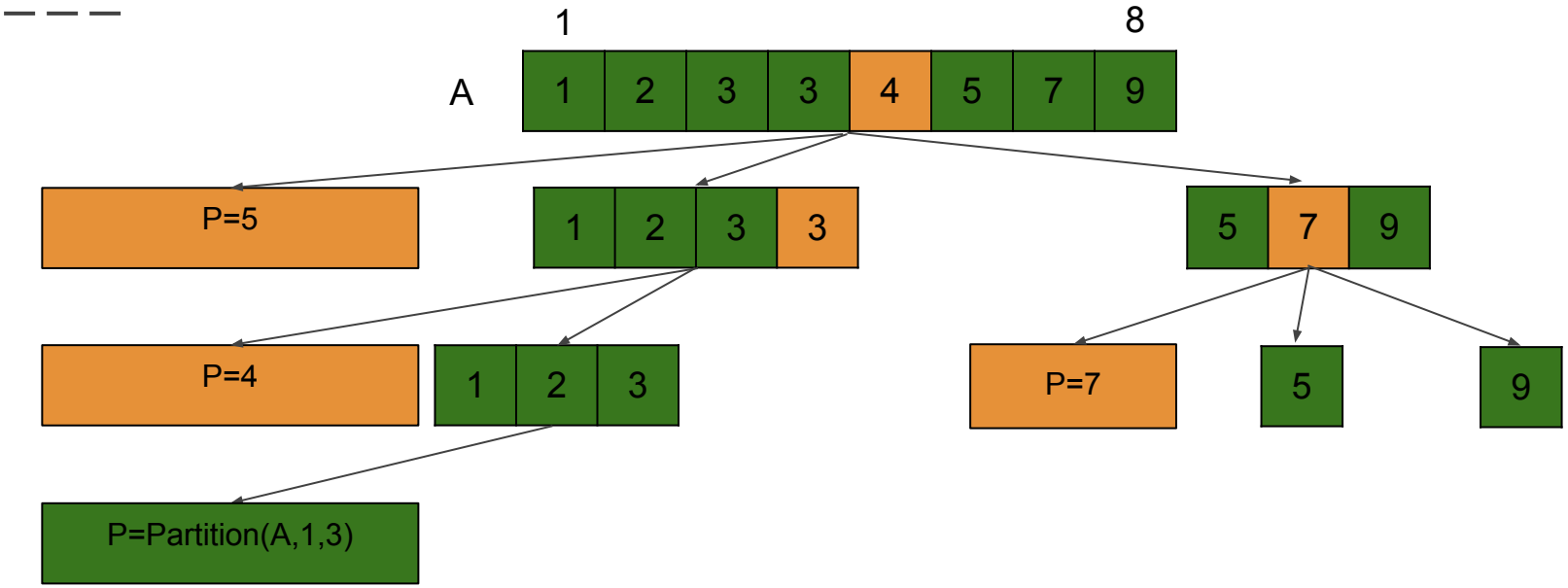
Simulation of Quick Sort



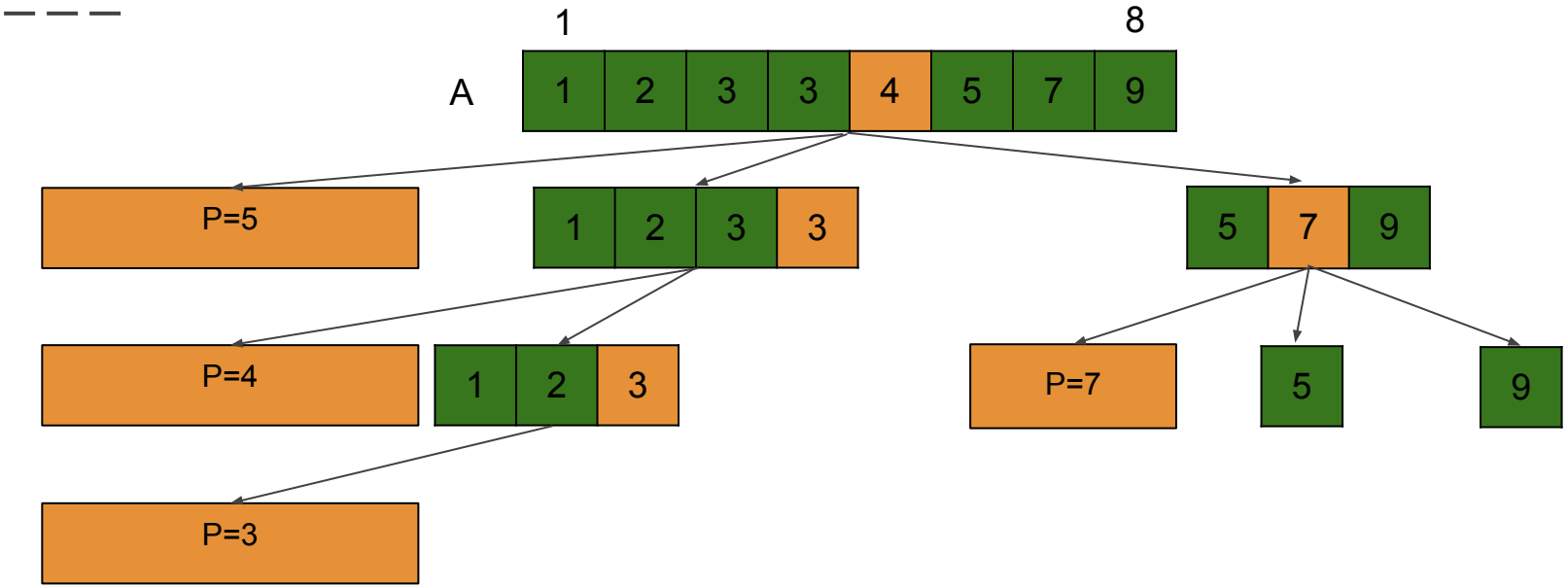
Simulation of Quick Sort



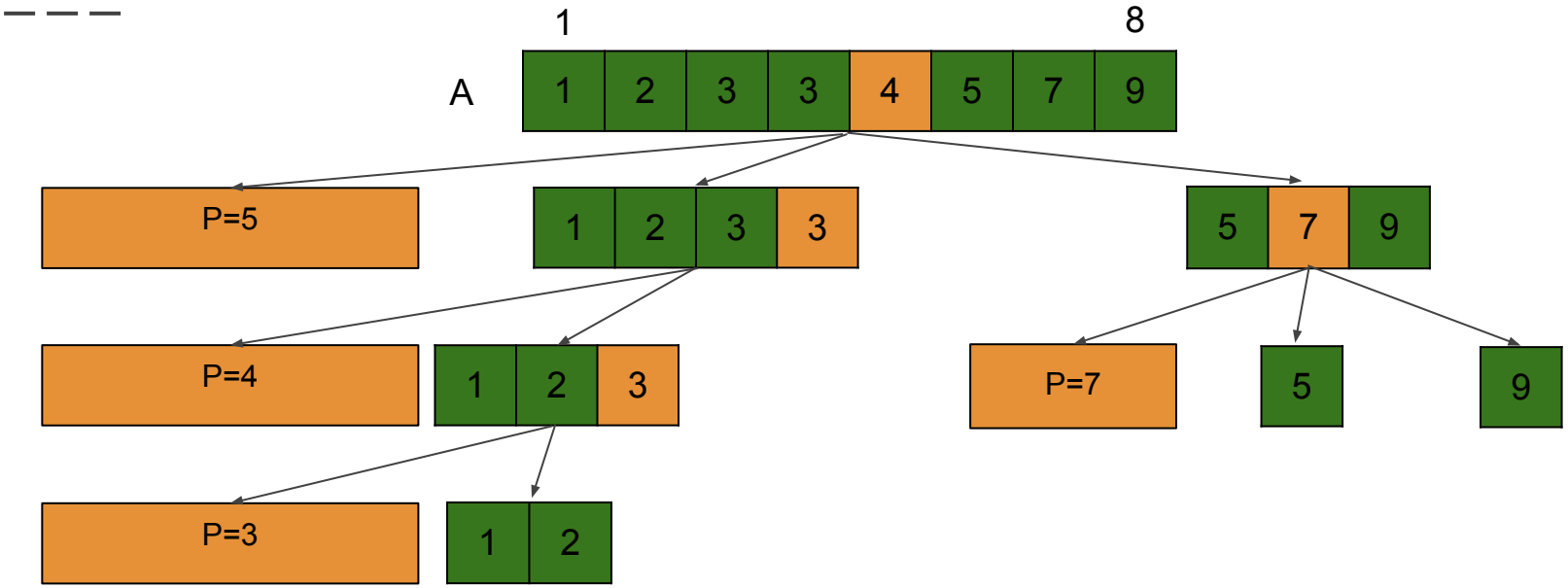
Simulation of Quick Sort



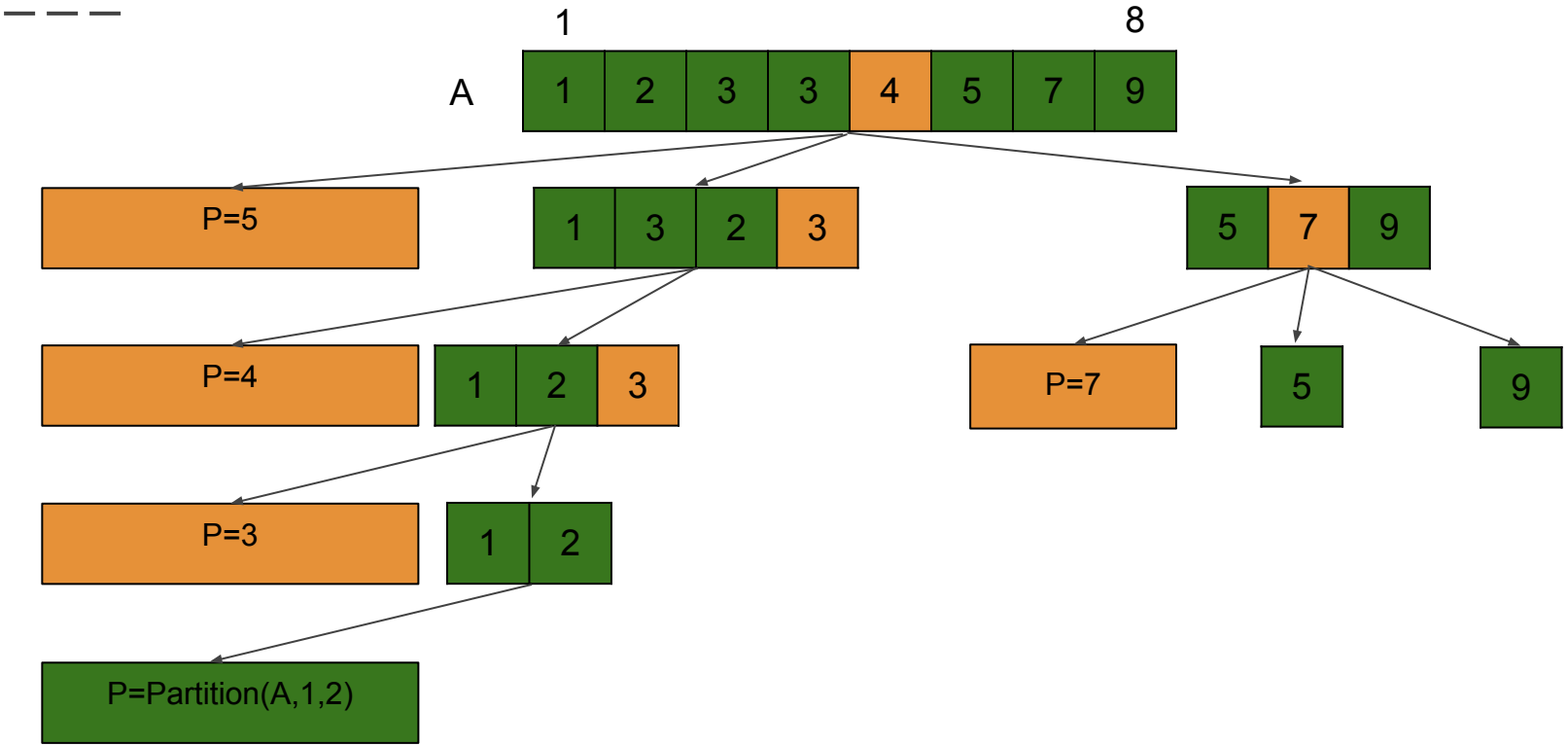
Simulation of Quick Sort



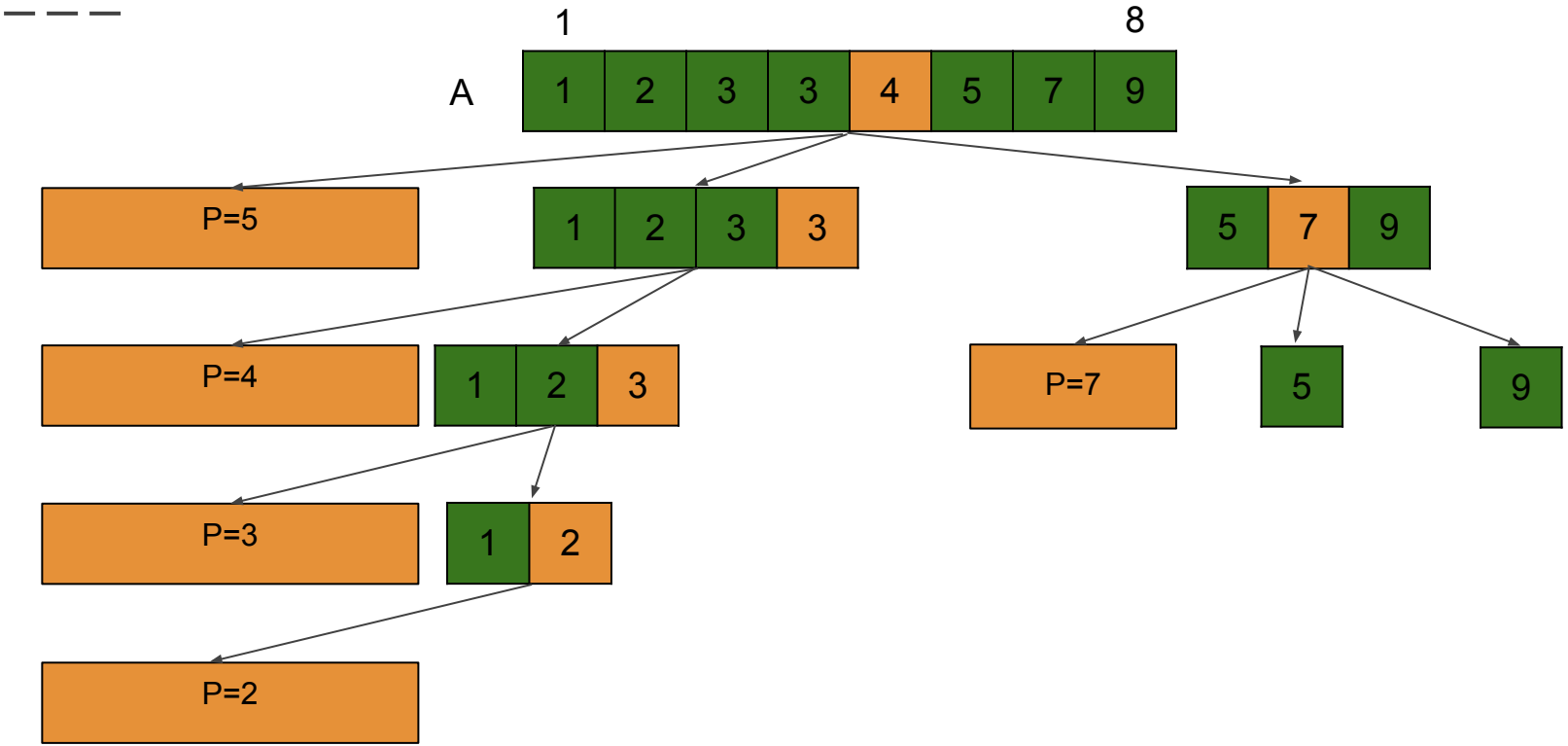
Simulation of Quick Sort



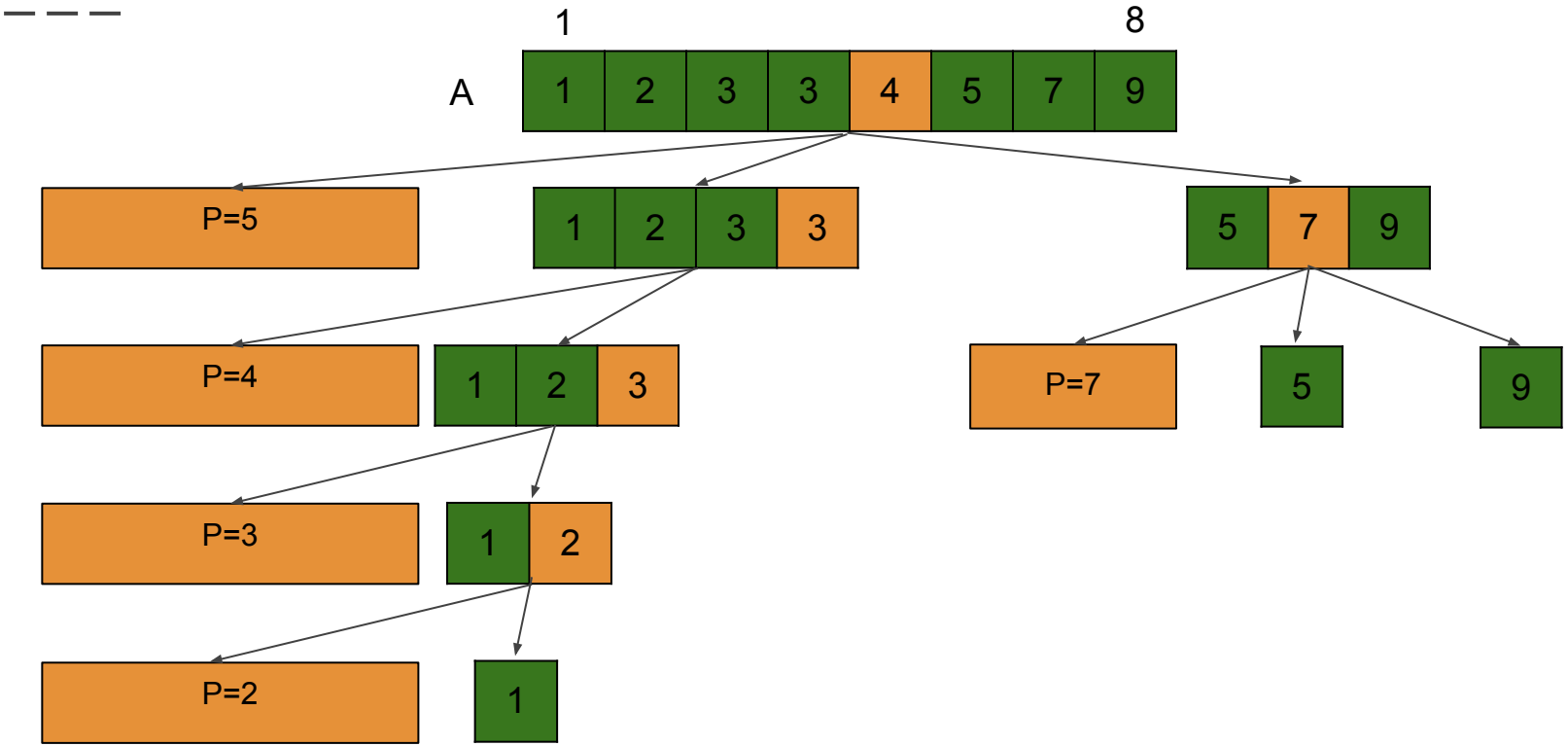
Simulation of Quick Sort



Simulation of Quick Sort



Simulation of Quick Sort



Analysis of Quick Sort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. In this section, we shall informally investigate how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

Analysis of Quick Sort [Worst Case] Continued

– – –

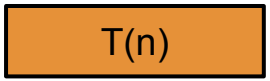
The worst case behavior for quicksort occurs when the partitioning routine produces one subproblem with n-1 elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs Θ(n) time. Since the recursive call on an array of size 0 just return T(0)=Θ(1), and the recurrence for the running time is ,

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(1) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

Analysis of Quick Sort [Worst Case] Continued

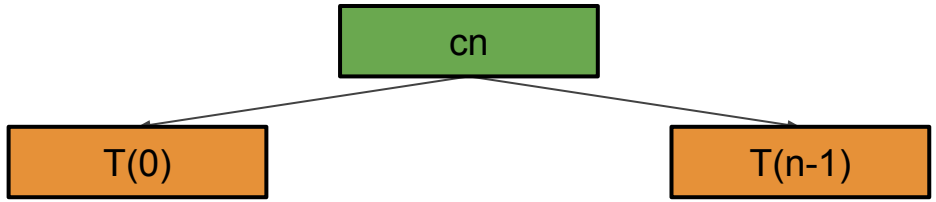
– – –

$T(n) = T(n-1) + \Theta(n)$



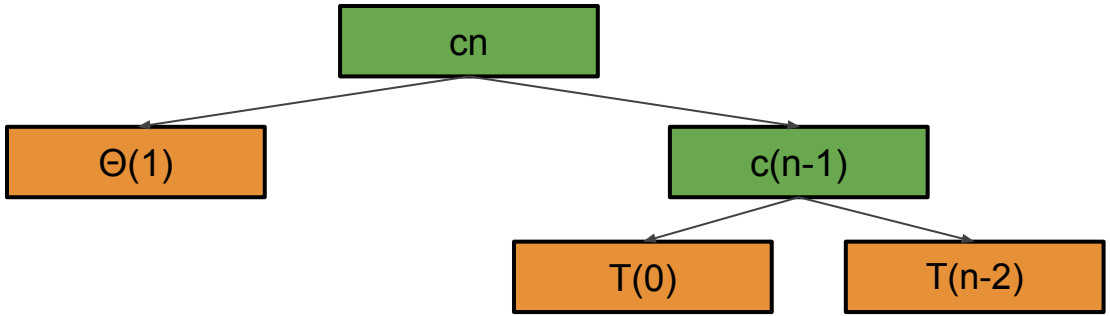
Analysis of Quick Sort [Worst Case] Continued

$T(n) = T(n-1) + \Theta(n)$



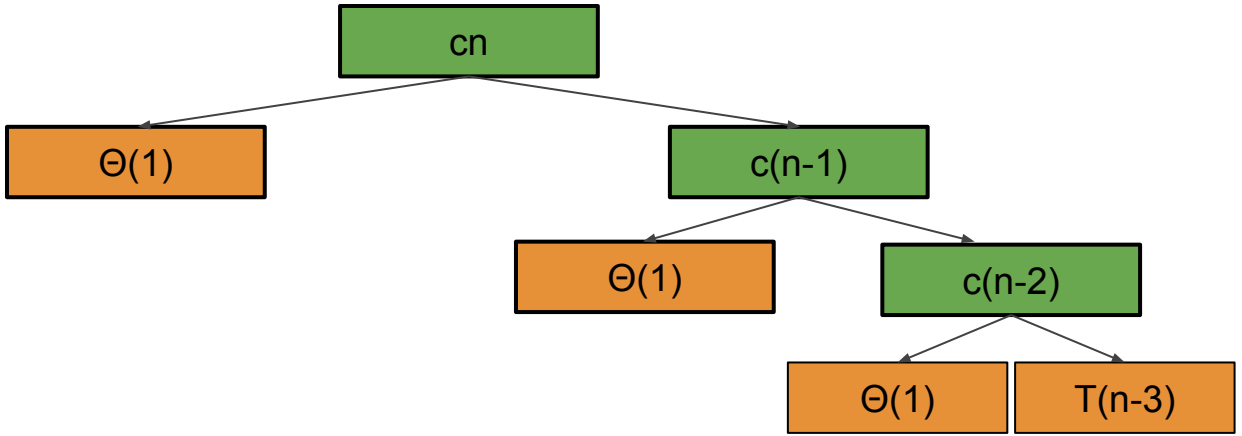
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



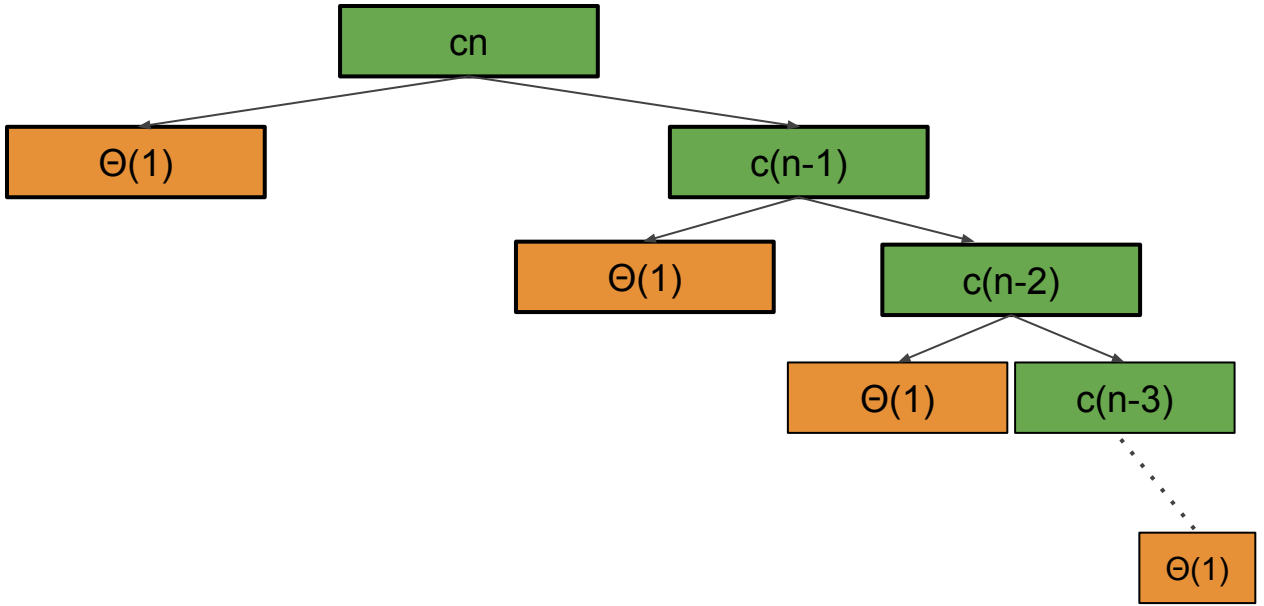
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



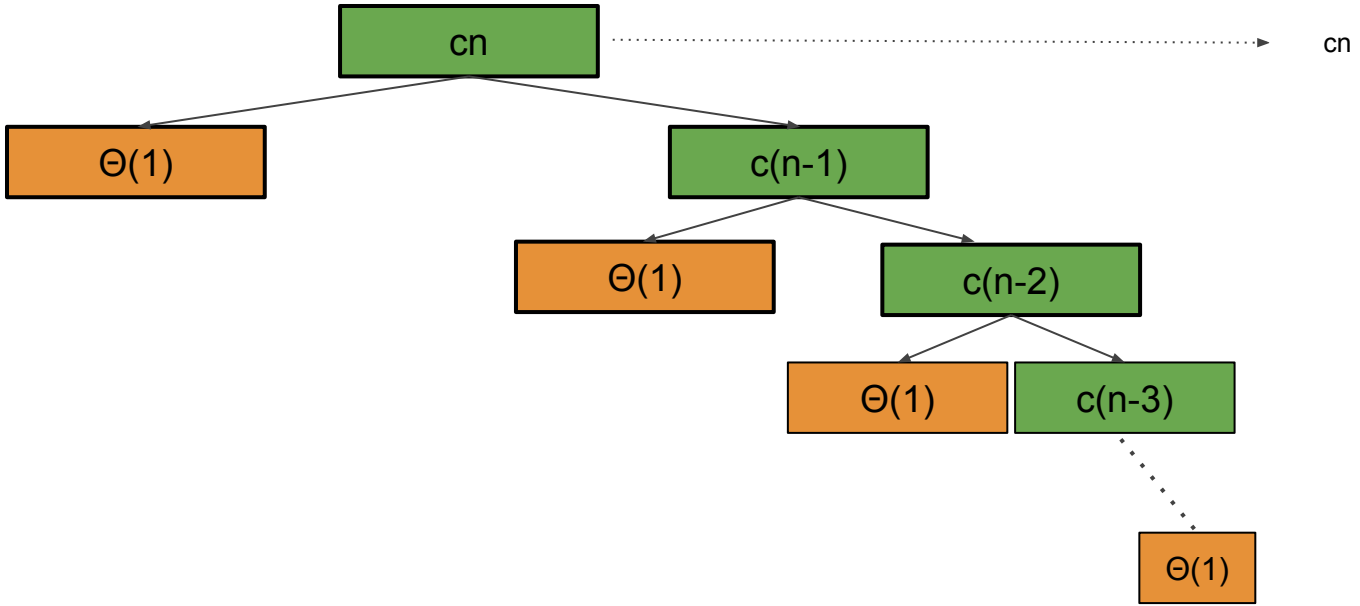
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



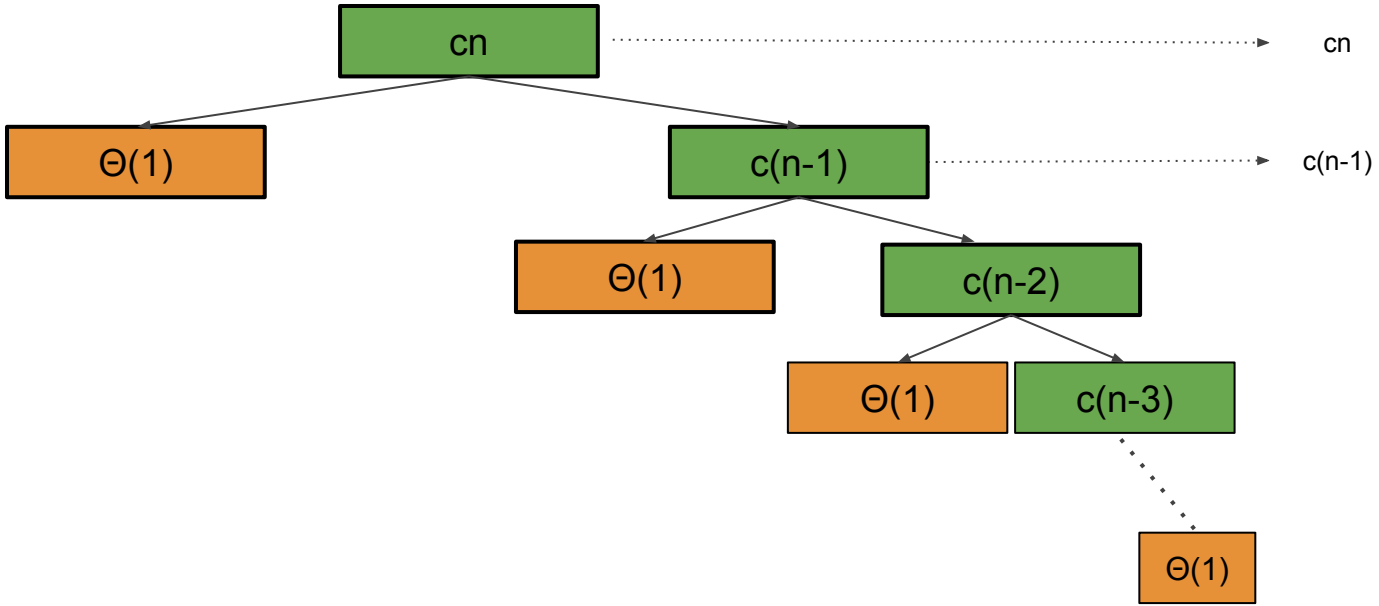
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



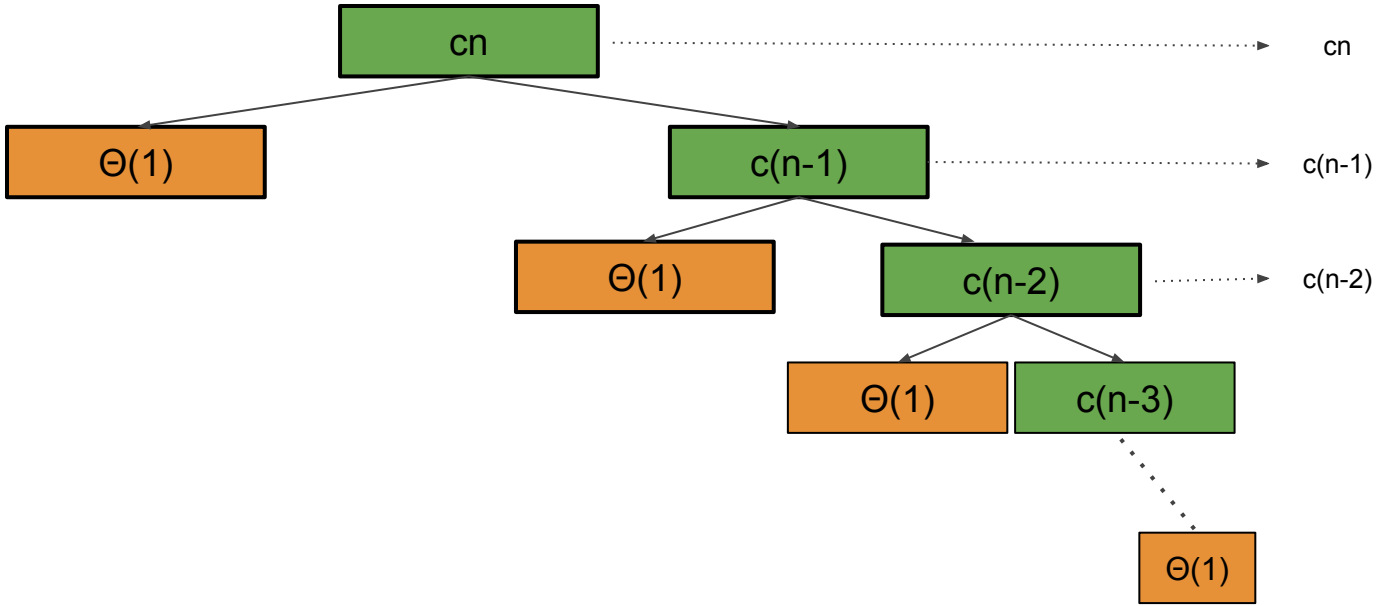
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



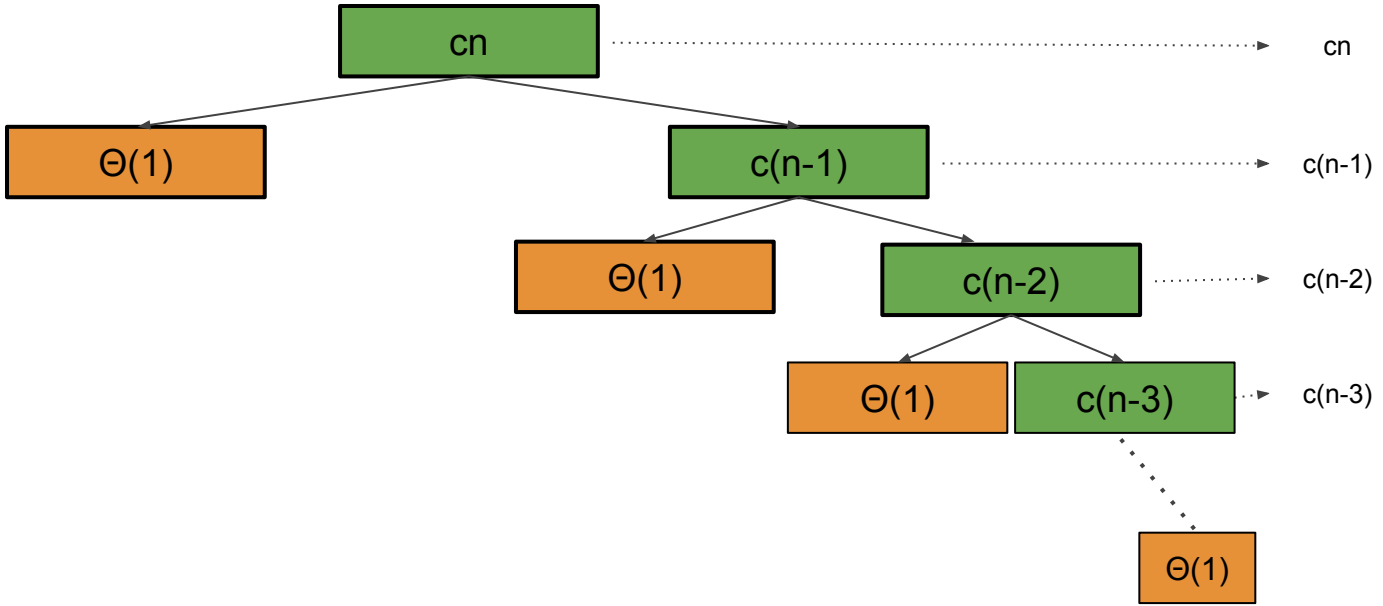
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



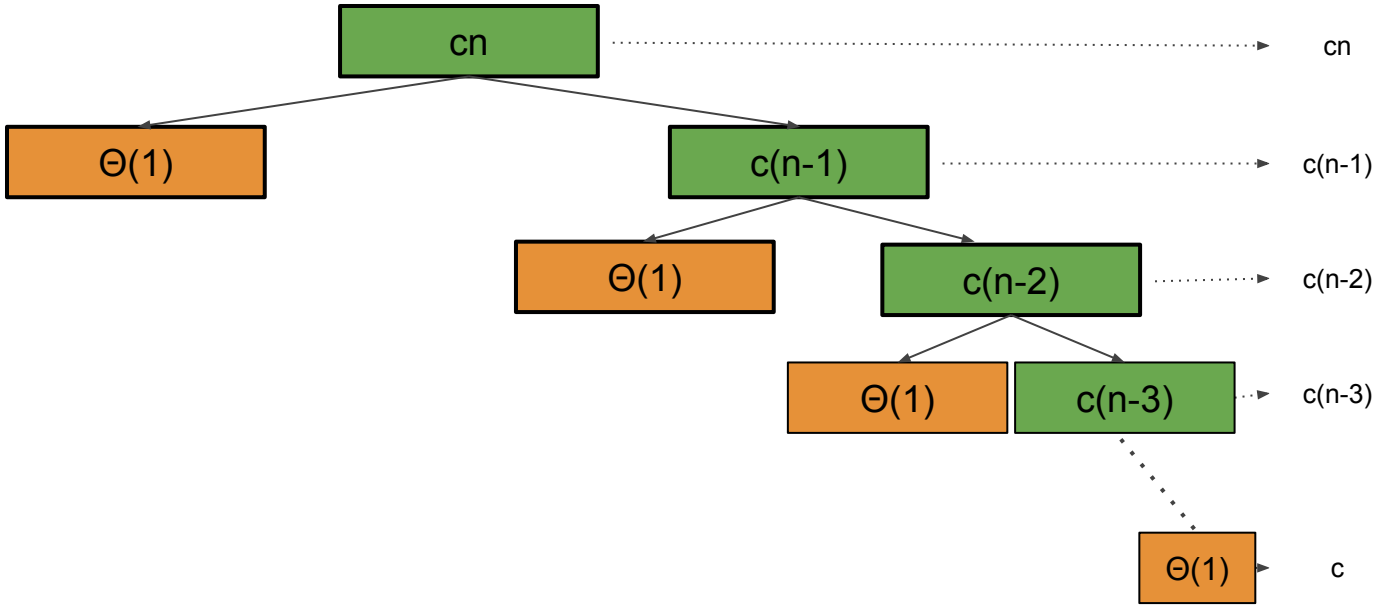
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



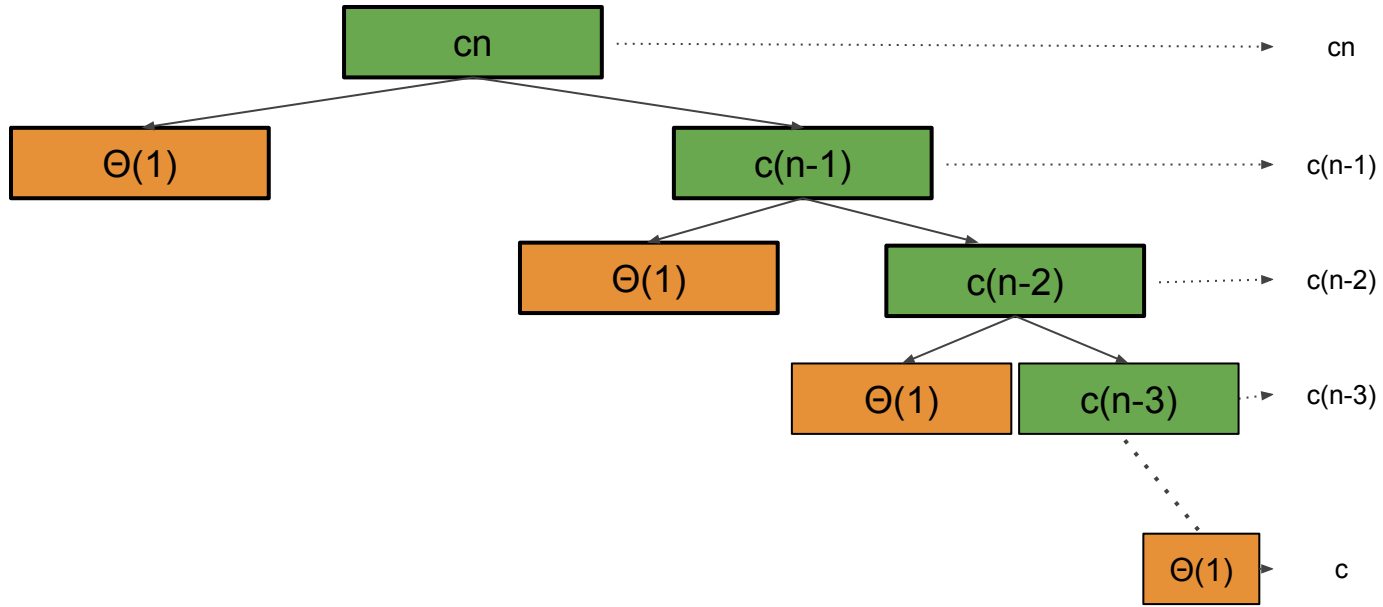
Analysis of Quick Sort [Worst Case] Continued

T(n) = T(n-1) + Θ(n)



Analysis of Quick Sort [Worst Case] Continued

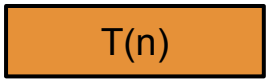
T(n) = T(n-1) + Θ(n)



T(n) = c(n + (n-1) + (n-2) + ... + 1)
= c(n(n+1)/2)
= **Θ(n^2)**

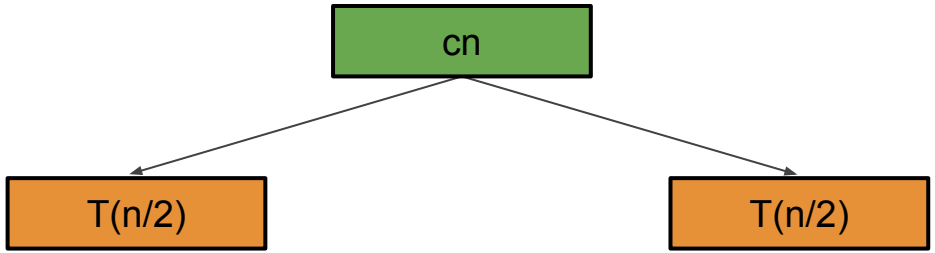
Analysis of Quick Sort [Best Case] Continued

$T(n) = 2T(n/2) + \Theta(n)$



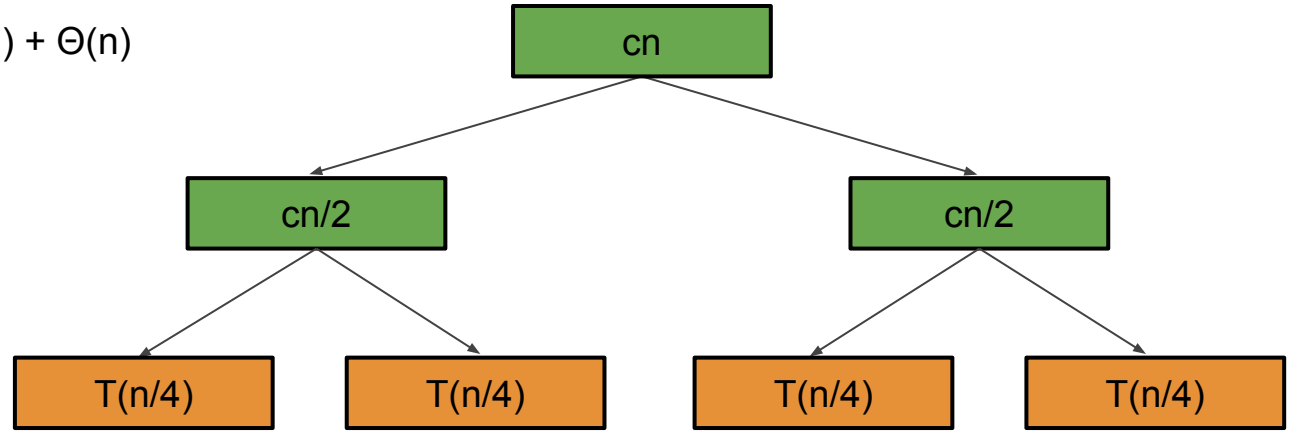
Analysis of Quick Sort [Best Case] Continued

$T(n) = 2T(n/2) + \Theta(n)$



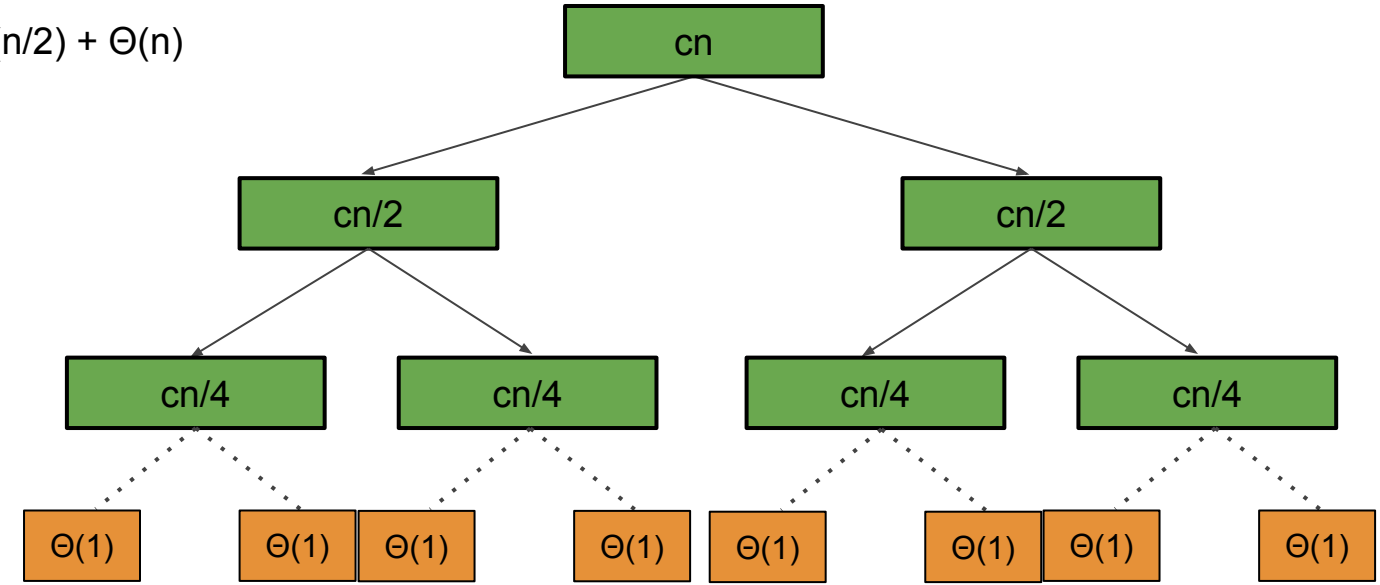
Analysis of Quick Sort [Best Case] Continued

$T(n) = 2T(n/2) + \Theta(n)$



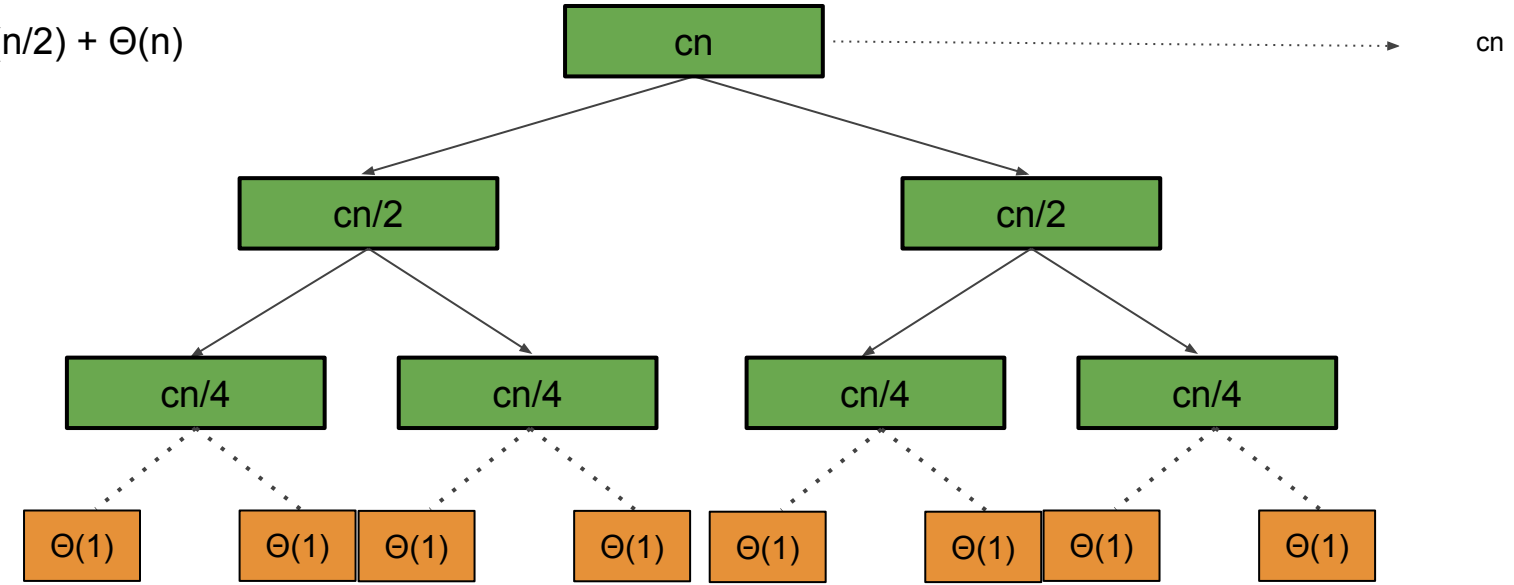
Analysis of Quick Sort [Best Case] Continued

$T(n) = 2T(n/2) + \Theta(n)$



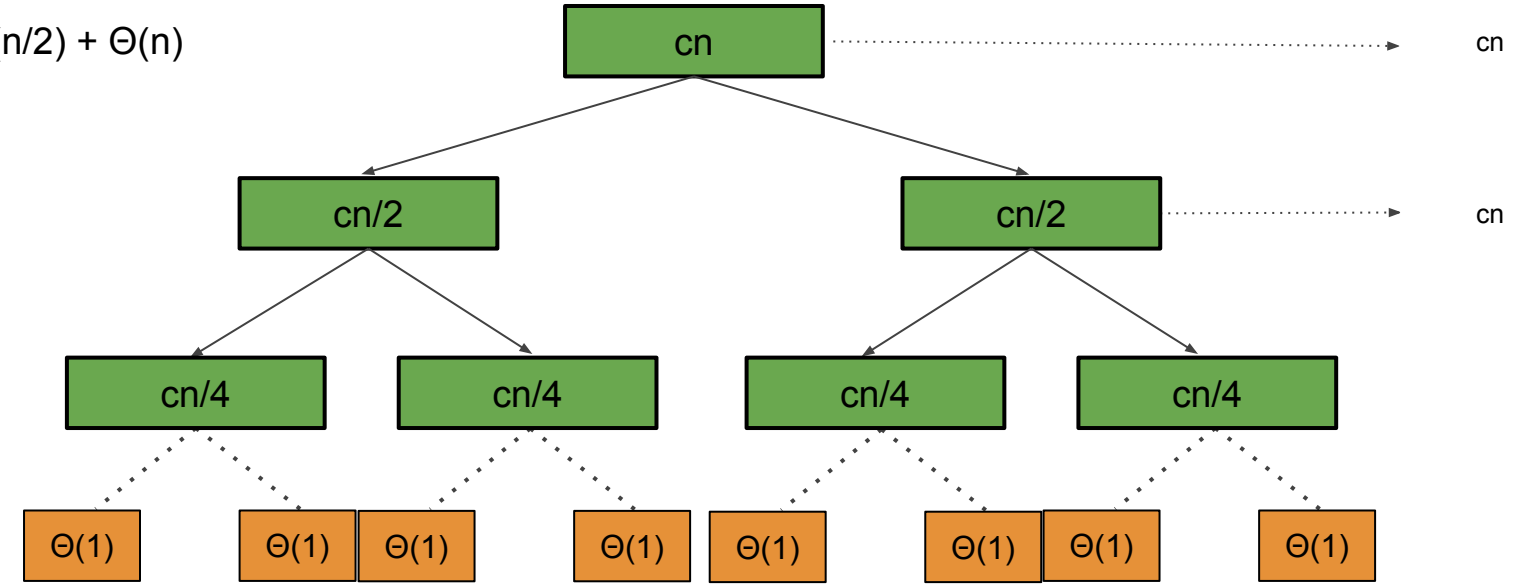
Analysis of Quick Sort [Best Case] Continued

$T(n) = 2T(n/2) + \Theta(n)$



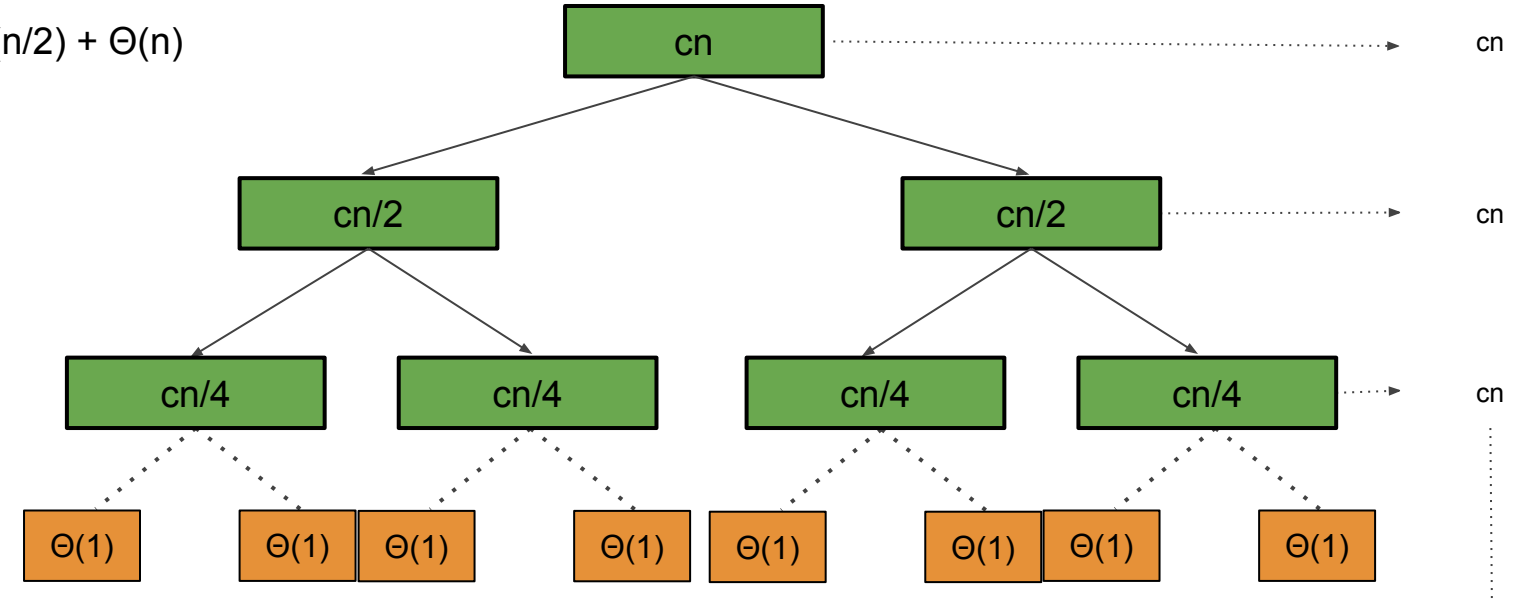
Analysis of Quick Sort [Best Case] Continued

$T(n) = 2T(n/2) + \Theta(n)$

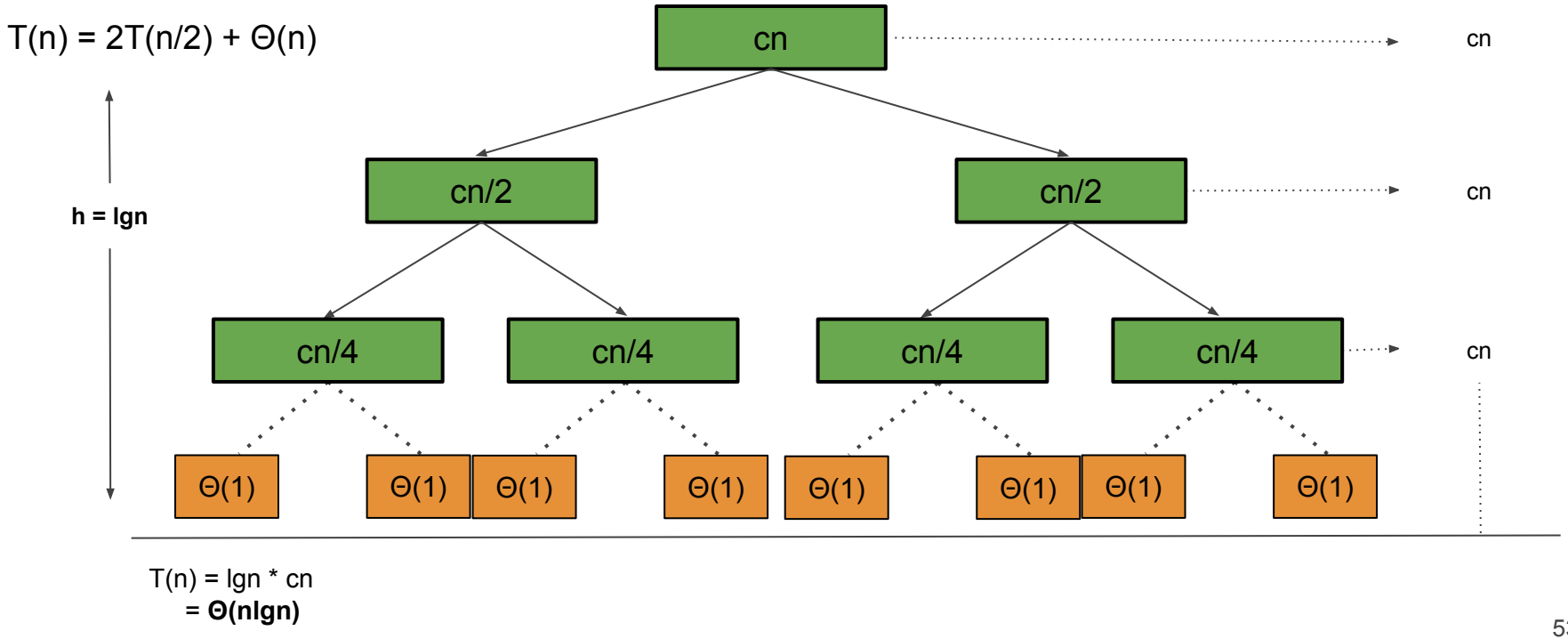


Analysis of Quick Sort [Best Case] Continued

$T(n) = 2T(n/2) + \Theta(n)$

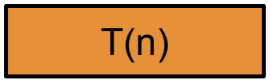


Analysis of Quick Sort [Best Case] Continued



Analysis of Quick Sort [Average Case] Continued – – –

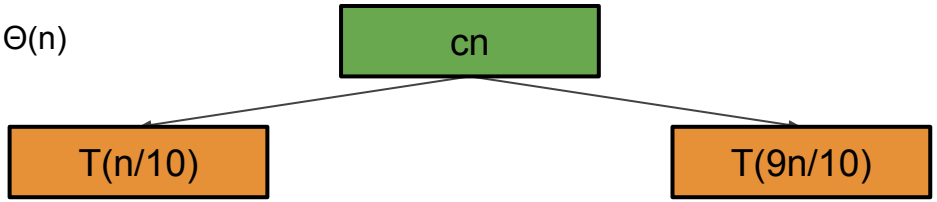
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

– – –

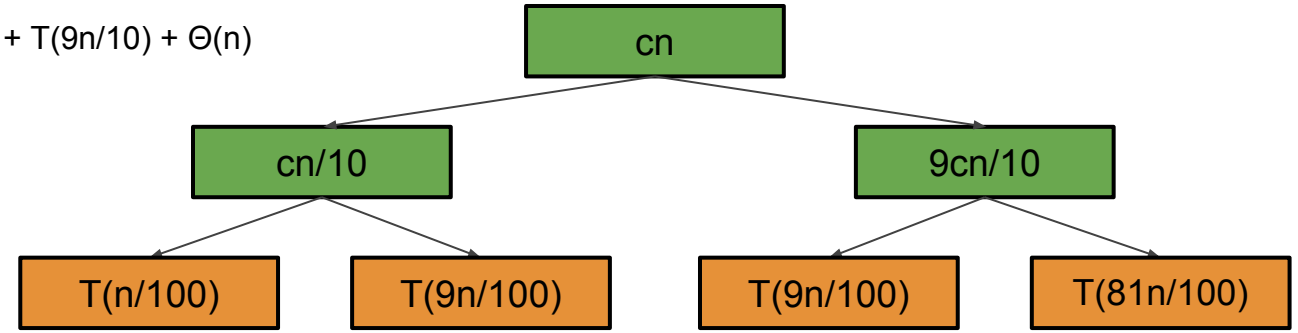
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

– – –

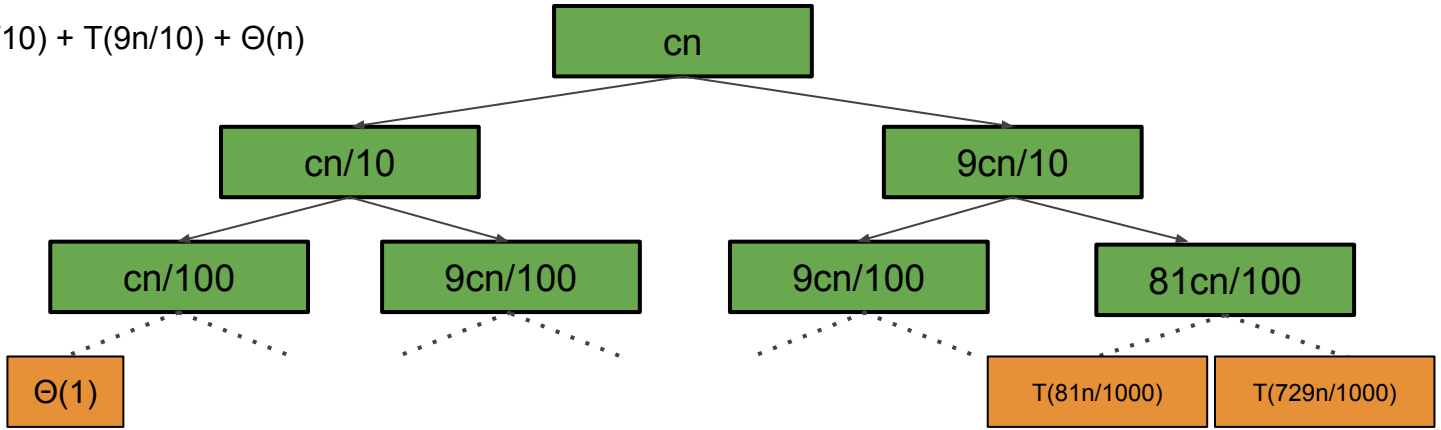
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

– – –

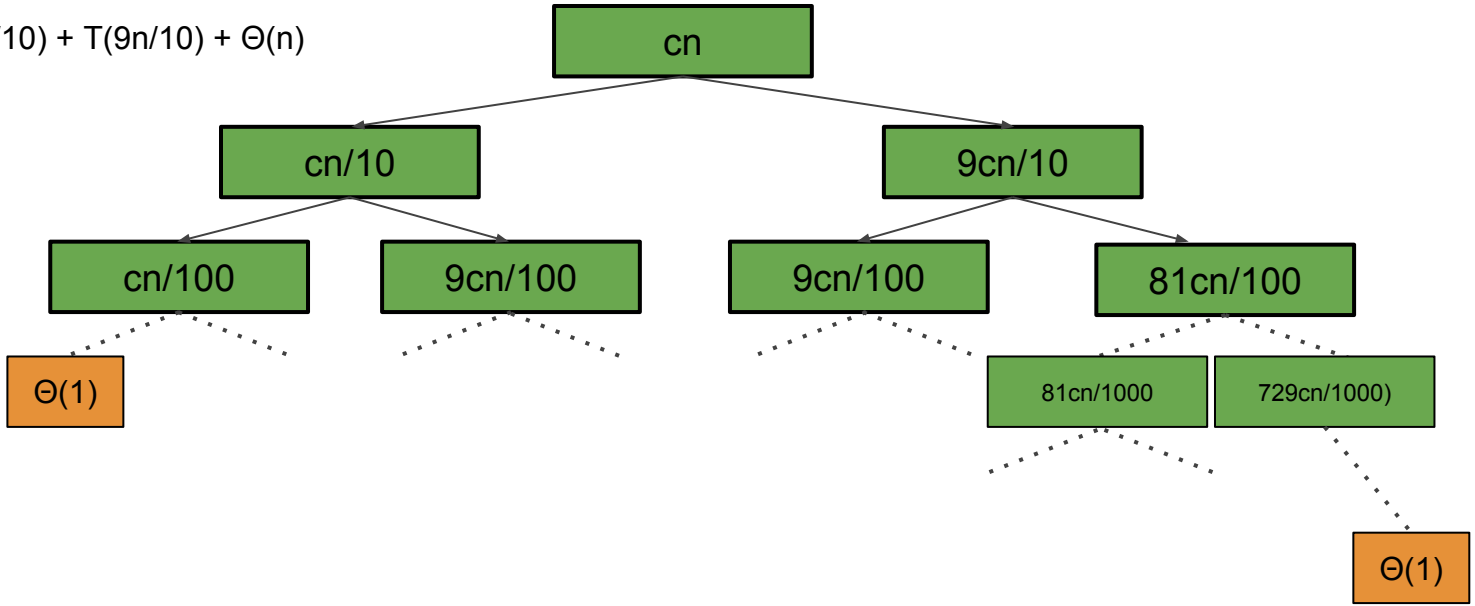
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

– – –

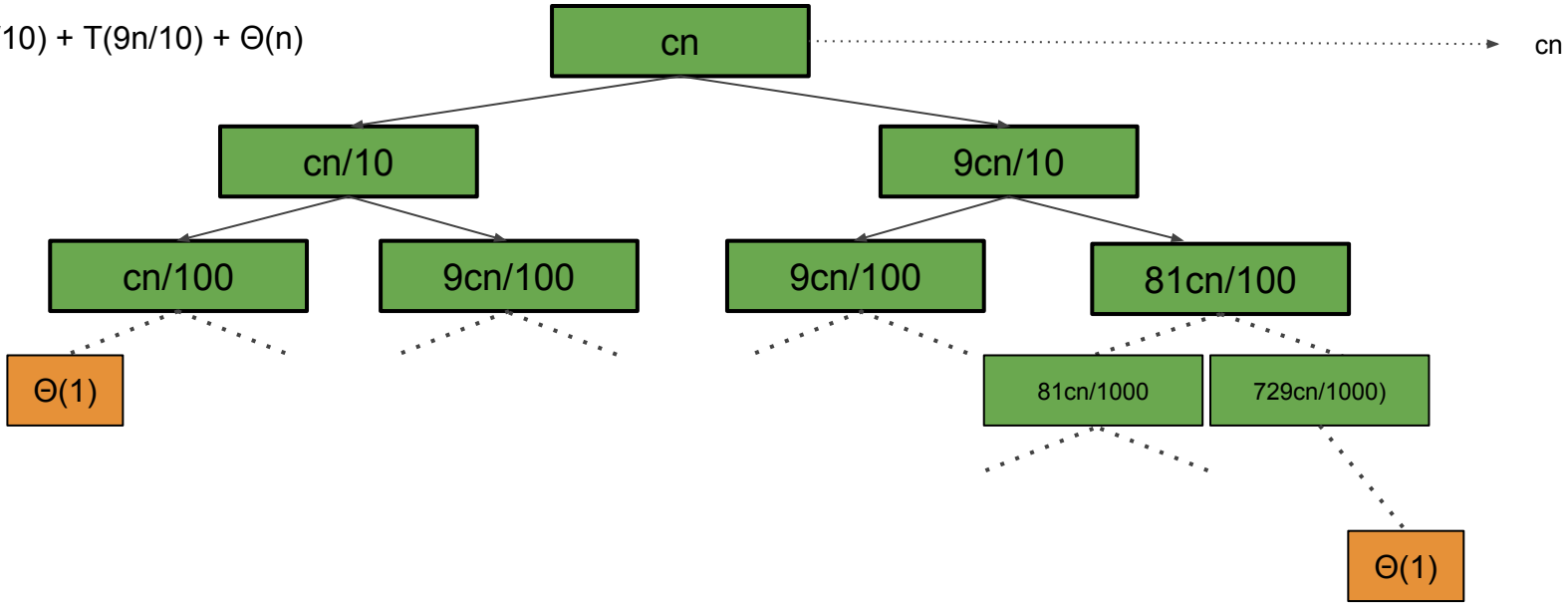
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

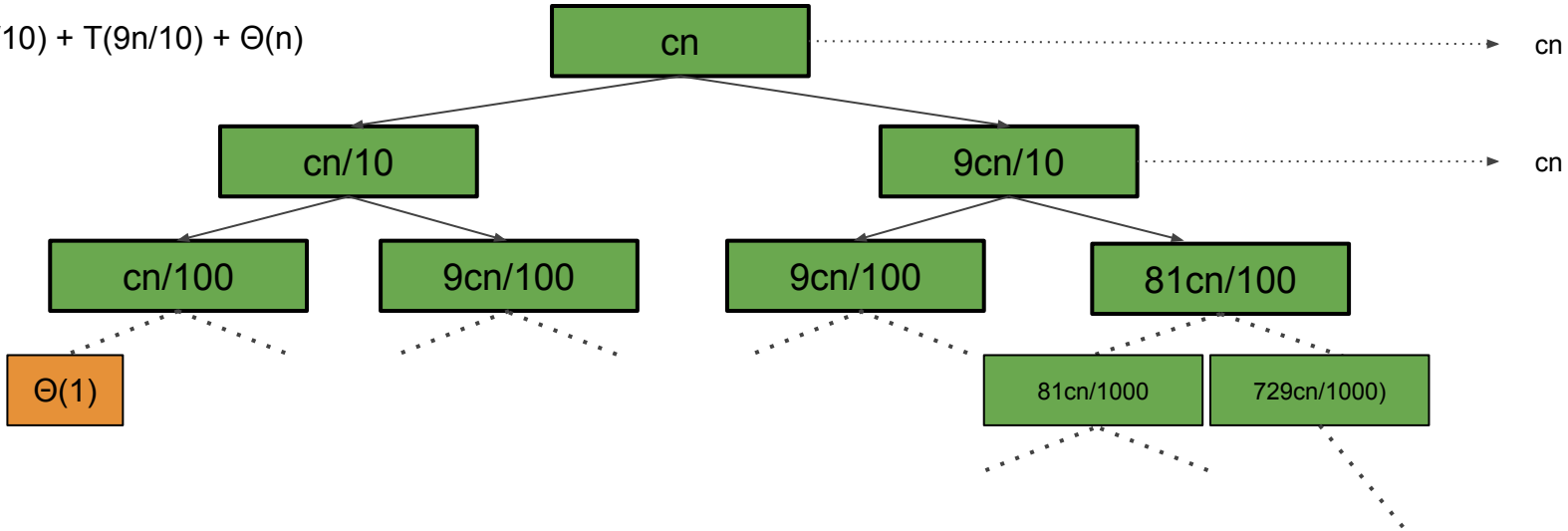
– – –

$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



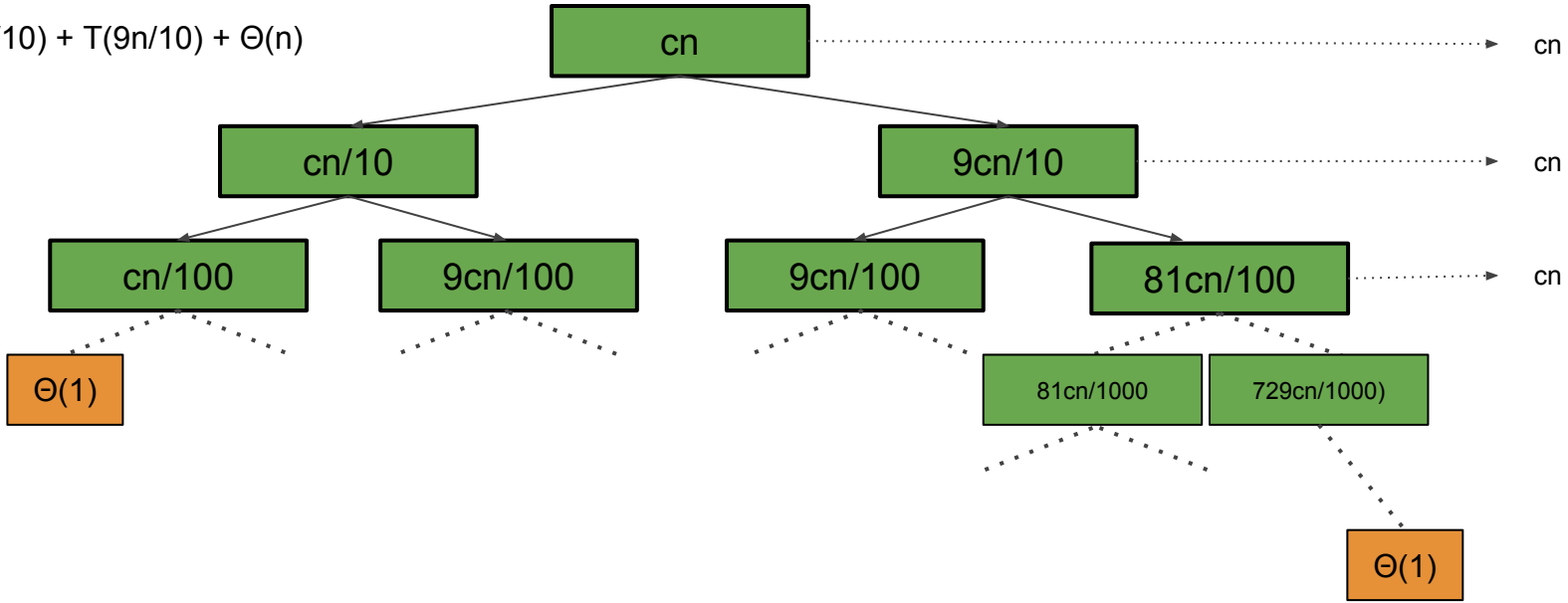
Analysis of Quick Sort [Average Case] Continued

$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

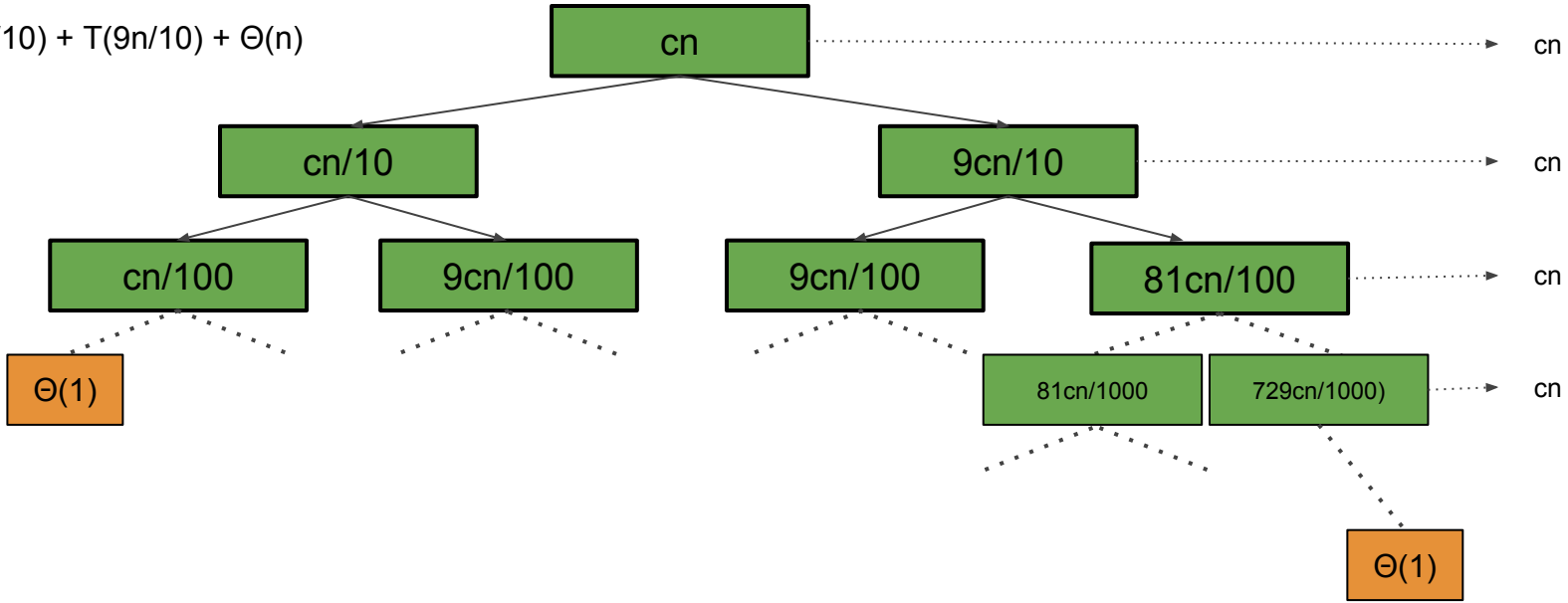
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

– – –

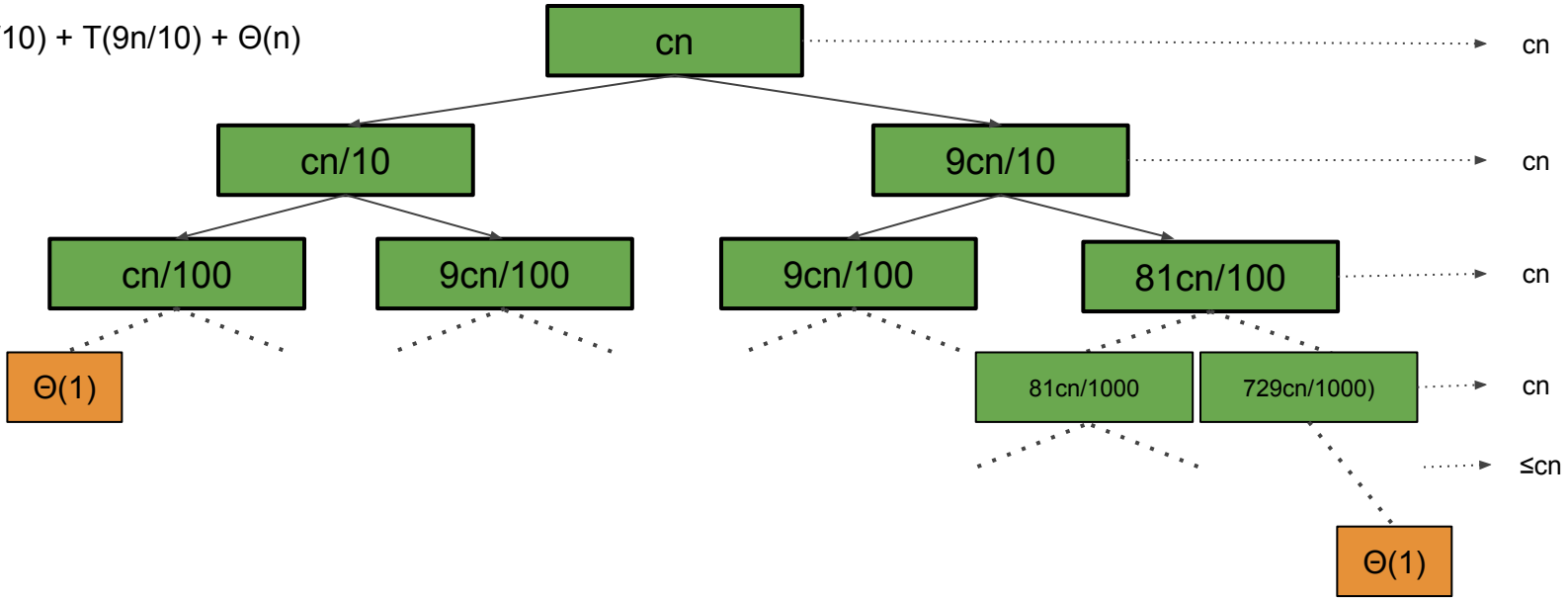
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued

– – –

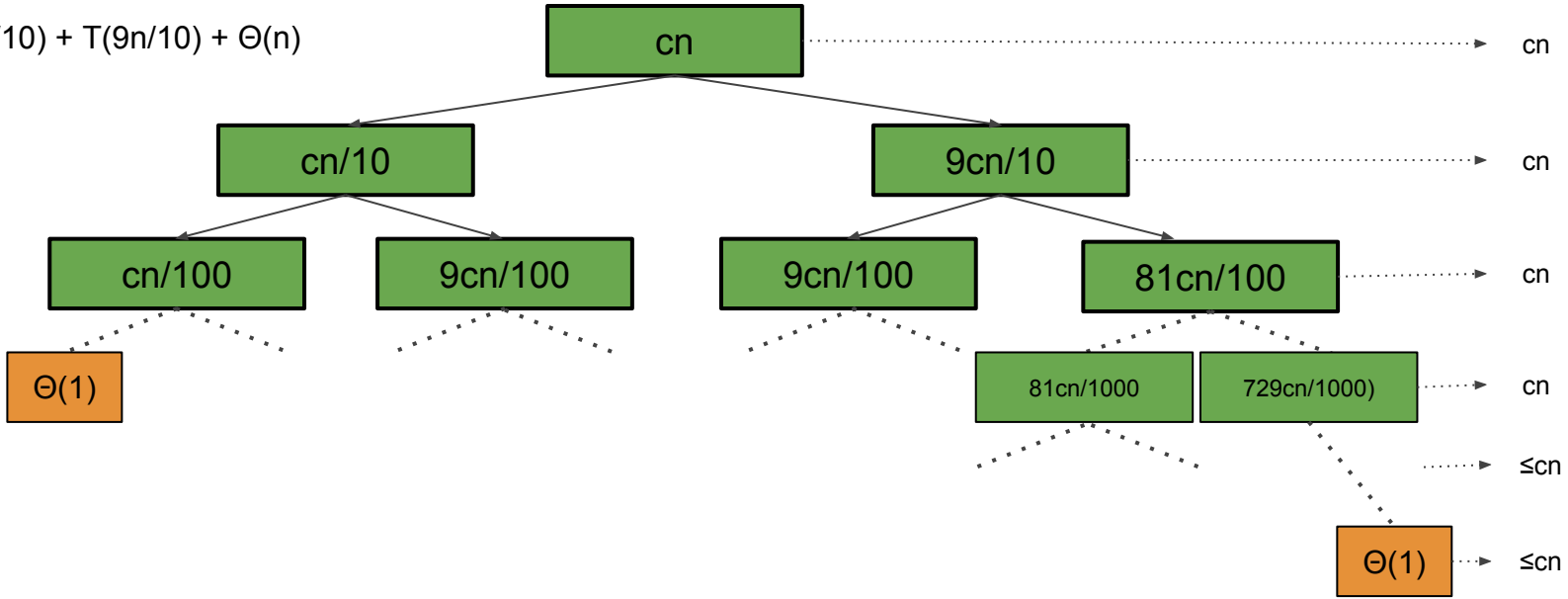
$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



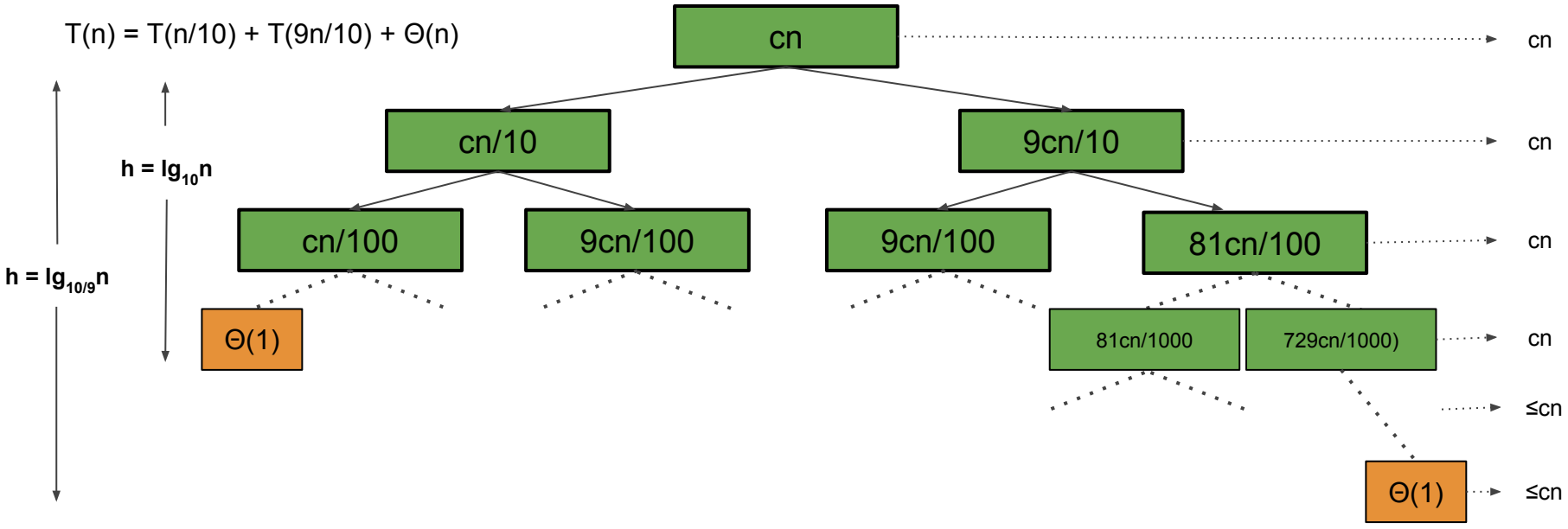
Analysis of Quick Sort [Average Case] Continued

– – –

$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



Analysis of Quick Sort [Average Case] Continued



Intuition for the Average Case

1. The behavior of quicksort depends on the relative ordering of the values in the array elements given as the input, and not by the particular values in the array.
2. We will assume for now that all permutations of the input numbers are equally likely.
3. When we run quicksort on a random input array, the partitioning is highly unlikely to happen in the same way at every level.
4. We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced.

Intuition for the Average Case Continued

– – –

5. In the average case, PARTITION produces a mix of “good” and “bad” splits.
6. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.

Suppose, for the sake of intuition, that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. The splits at two consecutive levels in the recursion tree.

Intuition for the Average Case Continued

– – –

At the root of the tree, the cost is n for partitioning, and the subarrays produced have sizes n - 1 and 0, the worst case. At the next level, the subarray of size n - 1 undergoes best-case partitioning into subarrays of size (n - 1)/2 - 1 and (n - 1)/2.

Let’s assume that the boundary-condition cost is 1 for the subarray of size 0.

Intuition for the Average Case Continued

– – –

Bad, $B(n) = G(n-1) + \Theta(n)$

Good, $G(n) = 2B(n/2) + \Theta(n)$

So now,

$G(n) = 2B(n/2) + \Theta(n)$

$= 2(G(n/2 - 1) + \Theta(n/2)) + \Theta(n)$

$= 2G(n/2 - 1) + \Theta(n)$

$= \Theta(\mathbf{nlgn})$

= Good

Randomized Quick Sort

1. In exploring the average-case behavior of quicksort, we have made an assumption that all permutations of the input numbers are equally likely.
2. In an engineering situation, however, we cannot always expect this assumption to hold. We can sometimes add randomization to an algorithm in order to obtain good expected performance over all inputs.
3. We randomized our algorithm by explicitly permuting the input. We could do so for quicksort also, but a different randomization technique, called random sampling, yields a simpler analysis. Instead of always using A[r] as the pivot, we will select a randomly chosen element from the subarray A[p...r].

Randomized Quick Sort Continued

– – –

4. We do so by first exchanging element A[r] with an element chosen at random from A[p...r].
5. By randomly sampling the range p...r, we ensure that the pivot element x = A[r] is equally likely to be any of the r - p + 1 elements in the subarray.
6. Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average.

Randomized Quick Sort Continued

– – –

Randomized_Partiton(A, p, r)

```
    i = Random(p, r)
    exchange A[r] with A[i]
    return Partition(A, p, r)
```

Randomized_QuickSort(A, p, r)

```
    if p<r:
        q = Randomized_Partiton(A, p, r)
        Randomized_QuickSort(A, p, q - 1)
        Randomized_QuickSort(A, q + 1, r)
```