

## Lecture 6: Quicksort

**Dr. Khalil Yousef**

**Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University**

Read Chapter 7 of *Introduction to Algorithms*

# Quicksort

---

Quicksort, like merge sort, is a divide-and-conquer algorithm for sorting a subarray  $A[p..r]$ .

**Divide:** Partition the subarray  $A[l..r]$  into two subarrays:  $A[l..q - 1]$  and  $A[(q + 1)..r]$  so that all the elements in the first subarray are less than or equal to  $A[q]$ , and all of the elements in the second are greater than  $A[q]$ . The value  $q$  is computed during the partition process.

**Conquer:** Recursively sort the two subarrays.

**Combine:** No work required as the subarrays are sorted in-place.

## The Quicksort Algorithm

---

```
QUICKSORT( $A, l, r$ )
1. if  $l < r$ 
2.   then  $q \leftarrow \text{PARTITION}(A, l, r)$ 
3.       QUICKSORT( $A, l, q - 1$ )
4.       QUICKSORT( $A, q + 1, r$ )
```

The initial call is  $\text{QUICKSORT}(A, 1, \text{length}[A])$ .

## The Partition Algorithm

---

The Partition Algorithm (in English):

- Pick a **pivot** element (we'll use the element at the right end).
- Pass over elements from left to right, swapping any element  $\leq$  the pivot to a growing region at the left end. Left end gets filled with the “small” elements.
- After the pass, the pivot gets swapped to the place just after the “small element” region.
- Return the final index of the pivot.

## The Partition Algorithm *continued*

---

PARTITION( $A, l, r$ )

1.  $pivot \leftarrow A[r]$
2.  $fill \leftarrow l - 1$
3. **for**  $j \leftarrow l$  **to**  $r - 1$
4.     **do if**  $A[j] \leq pivot$
5.         **then** exchange  $A[++fill] \leftrightarrow A[j]$
7. exchange  $A[++fill] \leftrightarrow A[r]$     $\triangleright$  place pivot element
8. **return**  $fill$

The running time of PARTITION is  $\Theta(n)$ , where  $n = r - l + 1$ .

**Partition Example:**  $A = \langle 7, 8, 2, 6, 5, 1, 3, 4 \rangle$

---

Draw behavior of *fill*, *pivot*, and *j* during *PARTITION*( $A, 1, 8$ ) on overhead.

## The Partition Algorithm *continued*

---

The correctness of this algorithm can be shown using a loop invariant:

At the beginning of each iteration of the **for** loop,  $A[l..r]$  contains the same elements it started with, possibly rearranged, and, for any array index  $k$ ,

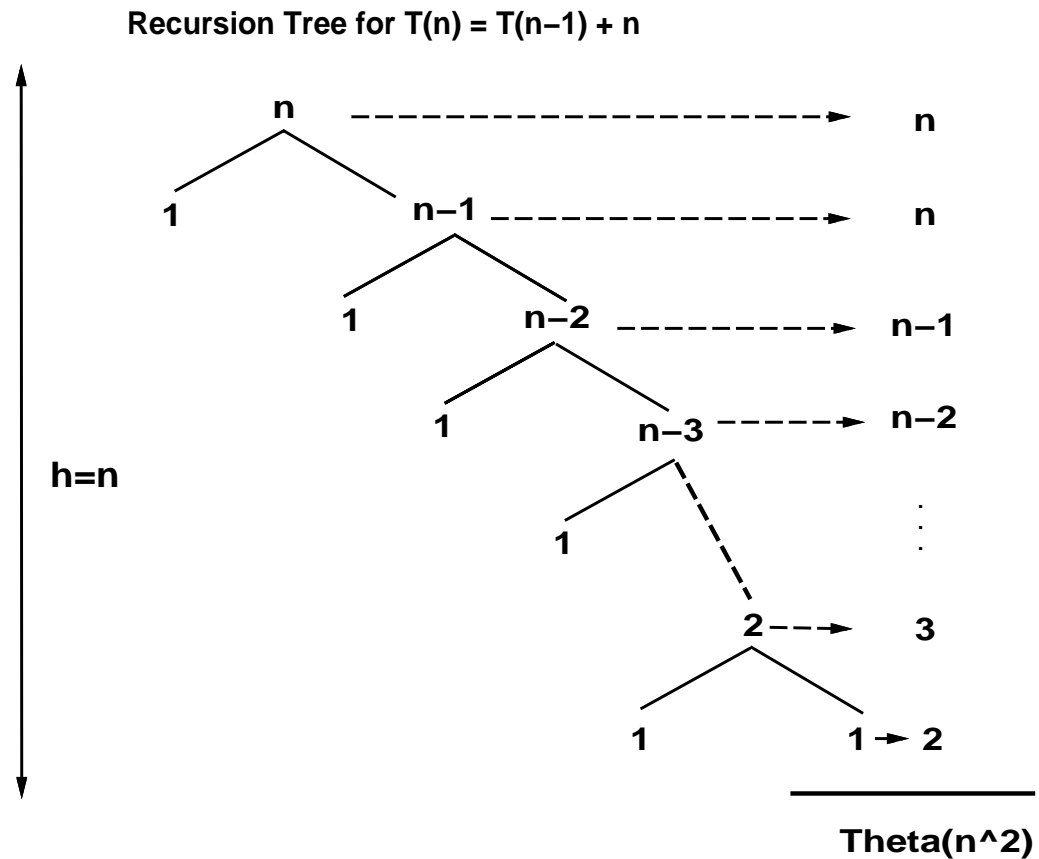
1. If  $l \leq k \leq i$ , then  $A[k] \leq pivot$ .
2. If  $i < k < j$ , then  $A[k] > pivot$ .
3. If  $k = r$ , then  $A[k] = pivot$ .

The running time of QUICKSORT depends on the sizes of the left and right partitions (which are affected by the choice of the pivot).

## Worst Case Performance of Quicksort

---

**Worst Case:** The left or right partition has size of one element.





## Worst Case Performance of Quicksort *continued*

---

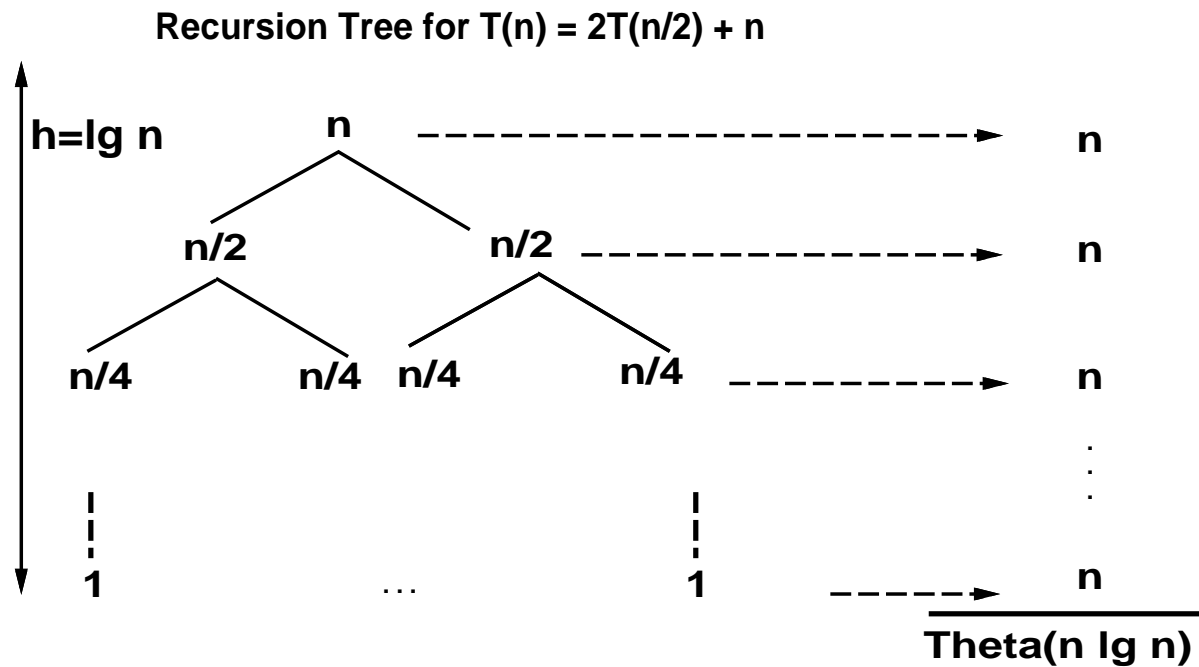
$$\begin{aligned}T(n) &= T(n-1) + \Theta(n) \\&= T(n-2) + \Theta(n-1) + \Theta(n) \\&= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\&\vdots \\&= \sum_{k=1}^n \Theta(k) \\&= \Theta\left(\sum_{k=1}^n k\right) \\&= \Theta(n^2)\end{aligned}$$

When does the worst case occur?

## Best Case Performance of Quicksort

---

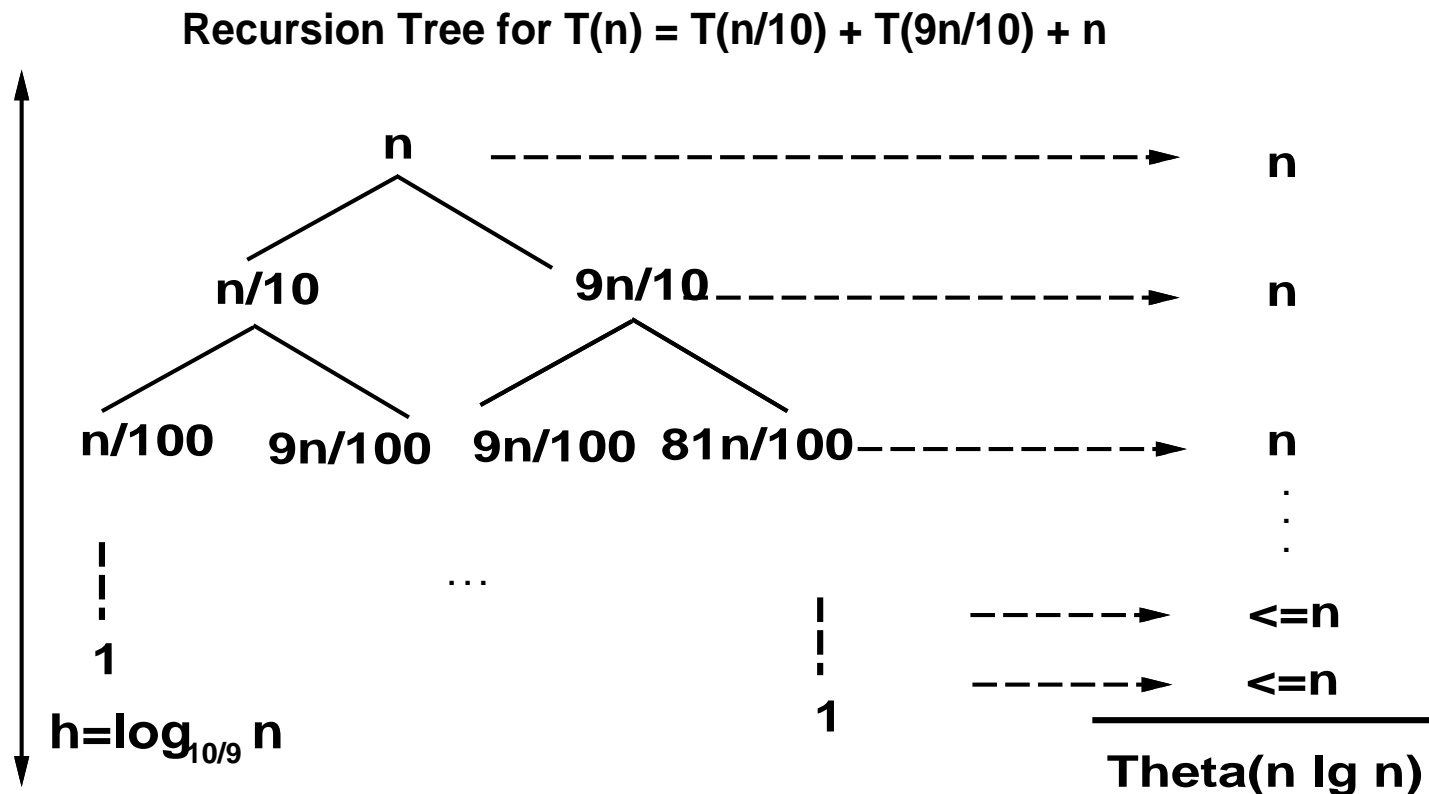
The best case occurs when the sizes of the left and right partitions are equal sized.



$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

## A Balanced Partition Case

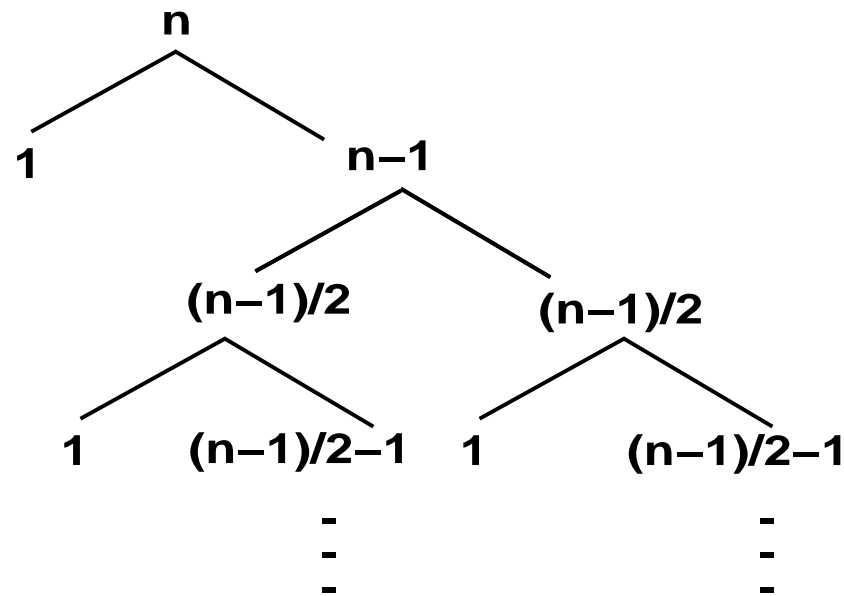
The average case running time of QUICKSORT is closer to the best case than the worst case. Even if there is an imbalance in the partitions of 9-to-1 (or 99-to-1 for that matter), we obtain a running time of  $\Theta(n \lg n)$ :



## Intuition for the Average Case

---

When we run QUICKSORT, it is unlikely that the partition occurs in the same way at every level; some will be balanced and some won't be. In the average case, PARTITION produces a random mix of “good” and “bad” splits. If the good and bad splits alternate over the levels, we end up with a nearly balanced partition which is better than the 9-to-1 case:



Average Case will be  $O(n \lg n)$

## Randomized Quicksort

---

QUICKSORT's performance depends on the distribution of the input data. If we create a randomized version of QUICKSORT, the worst case behavior will rely on the random number generator hitting an unlucky and unlikely partition, again and again.

One possible implementation of RANDOMIZED-QUICKSORT is to randomize PARTITION by picking the pivot element at random from the range  $l..r$ , exchange  $A[r]$  with that element, and run PARTITION as before.

**RANDOMIZED-PARTITION**( $A, l, r$ )

1.  $i \leftarrow \text{RANDOM}(l, r)$
2. exchange  $A[r] \leftrightarrow A[i]$
3. **return** PARTITION( $A, l, r$ )

## Randomized Quicksort *continued*

---

RANDOMIZED-QUICKSORT( $A, l, r$ )

1. **if**  $l < r$
2.     **then**  $q \leftarrow \text{RANDOMIZED-PARTITION}(A, l, r)$
3.         RANDOMIZED-QUICKSORT( $A, l, q - 1$ )
4.         RANDOMIZED-QUICKSORT( $A, q + 1, r$ )

**Note:** This does not improve the worst case running time of the algorithm, but it does prevent particular input cases from causing the worst-case behavior.

Thus the worst case running time is still  $O(n^2)$ .