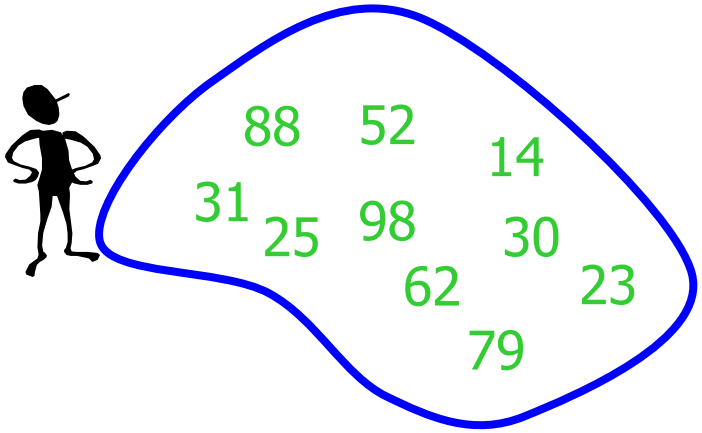
Design and Analysis of Algorithms

Quick Sort

Fall 2022

National University of Computer and Emerging Sciences, Islamabad

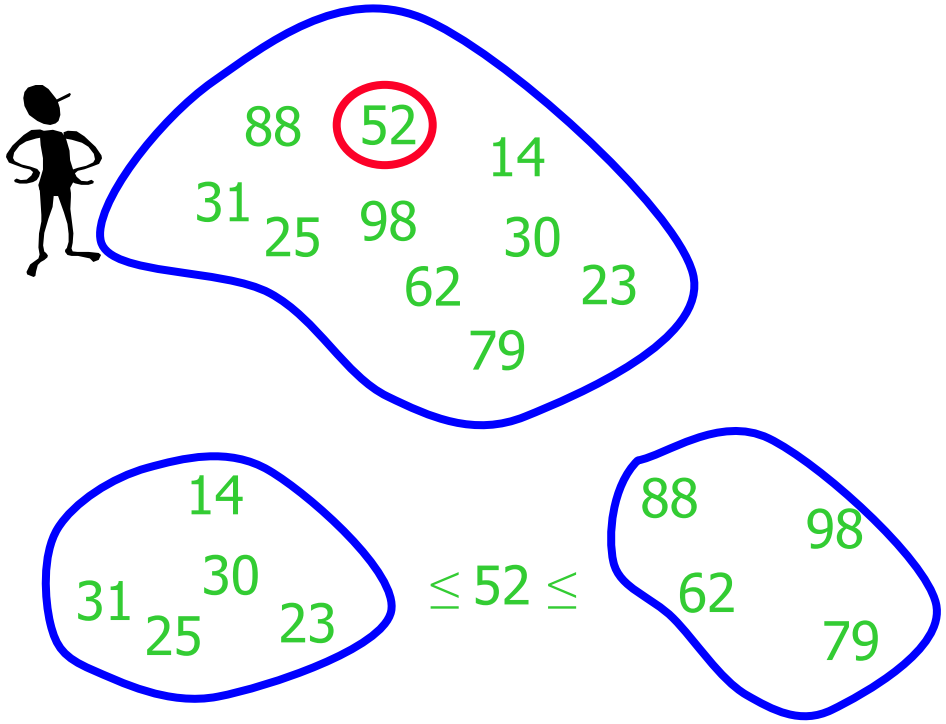
Quick Sort



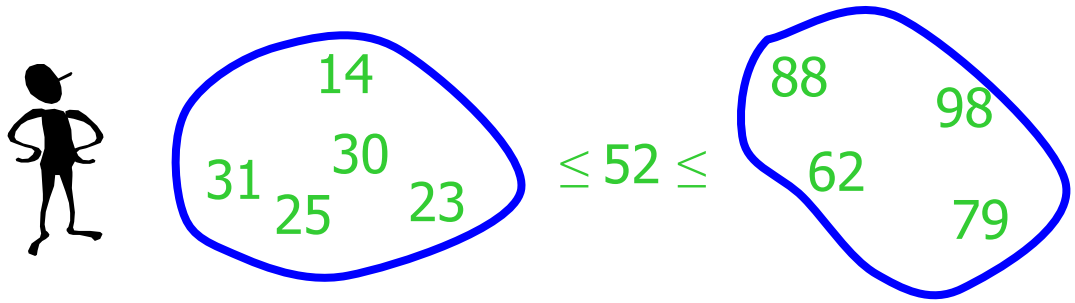
Divide and Conquer

Quick Sort

Partition set into two using randomly chosen pivot



Quick Sort



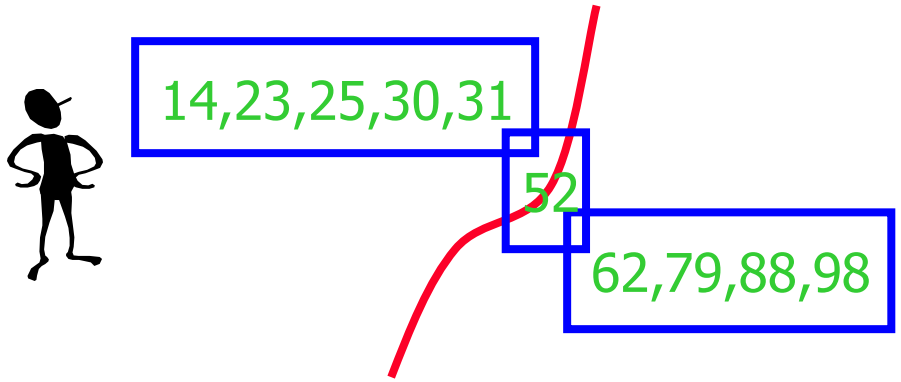
sort the first half.

14,23,25,30,31

sort the second half.

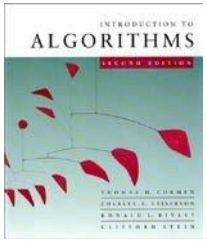
62,79,98,88

Quick Sort



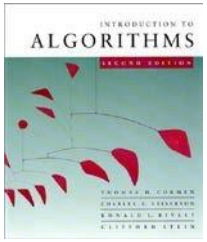
Glue pieces together.

14,23,25,30,31,52,62,79,88,98



Quicksort

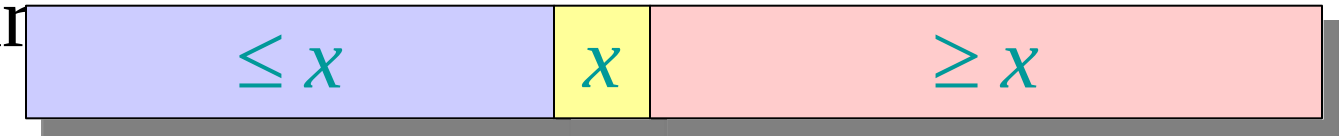
- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).



Divide and conquer

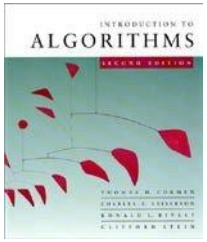
Quicksort an n-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper subar



2. ***Conquer:*** Recursively sort the two subarrays.
3. ***Combine:*** Trivial.

Key: *Linear-time partitioning subroutine.*

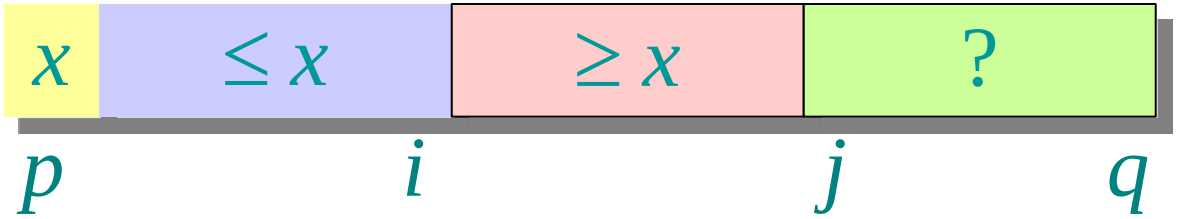


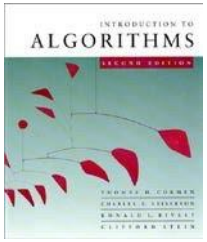
Partitioning subroutine

```
PARTITION(A, p, q)   ▷ A[ p . . q]
    x ← A[ p]        ▷ pivot = A[ p]
    i ← p
    for j ← p + 1 to q
        do if A[ j] ≤ x
                then  i ← i + 1
                      exchange A[i] ↔ A[ j]
    exchange A[ p] ↔ A[i]
    return i
```

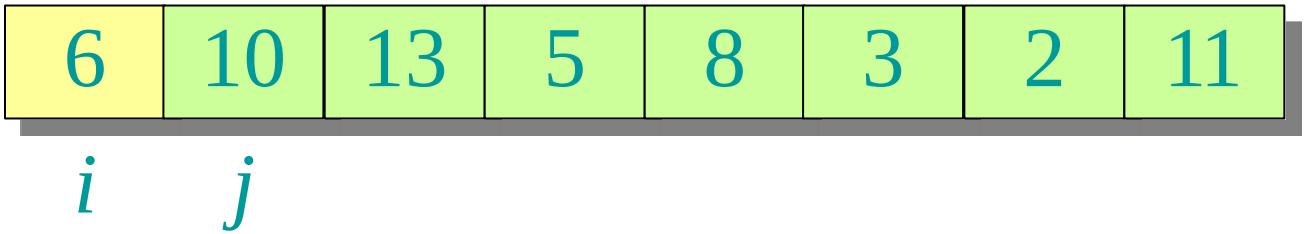
Running time = $O(n)$ for n elements.

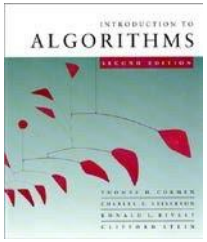
Invariant:



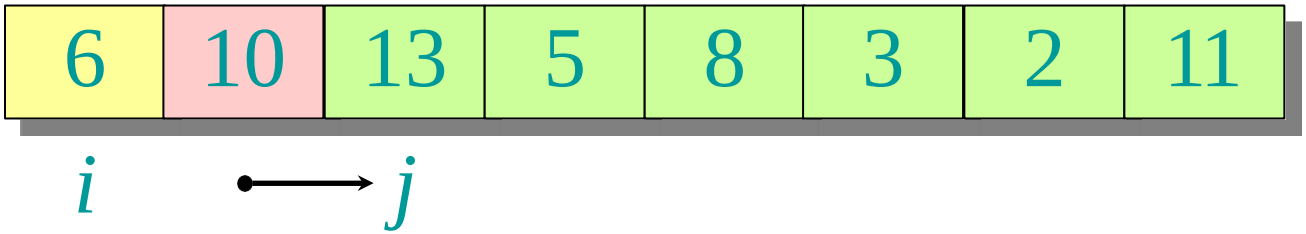


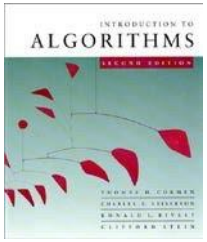
Example of partitioning



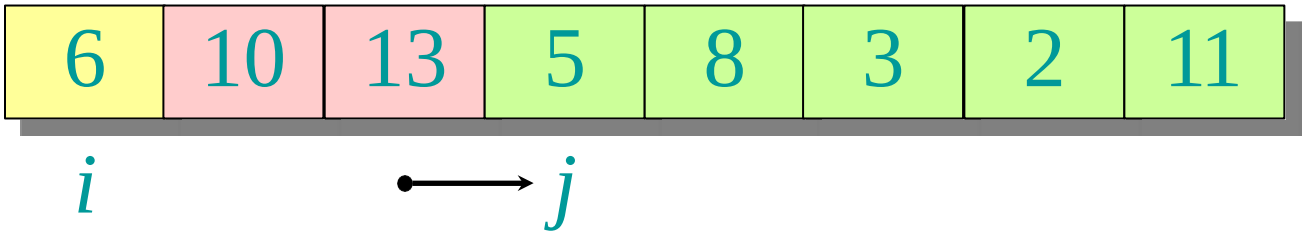


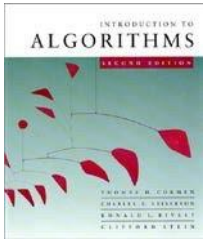
Example of partitioning



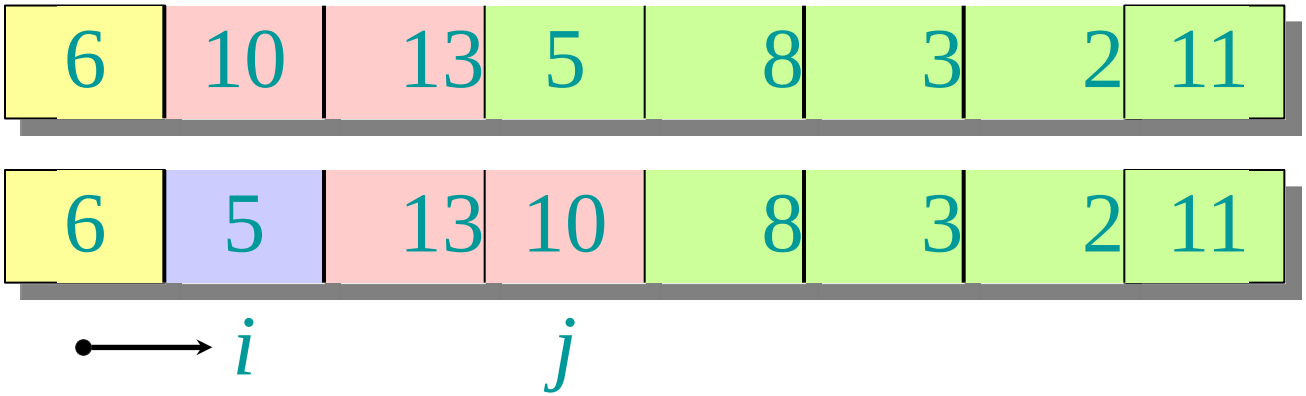


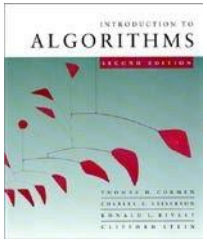
Example of partitioning



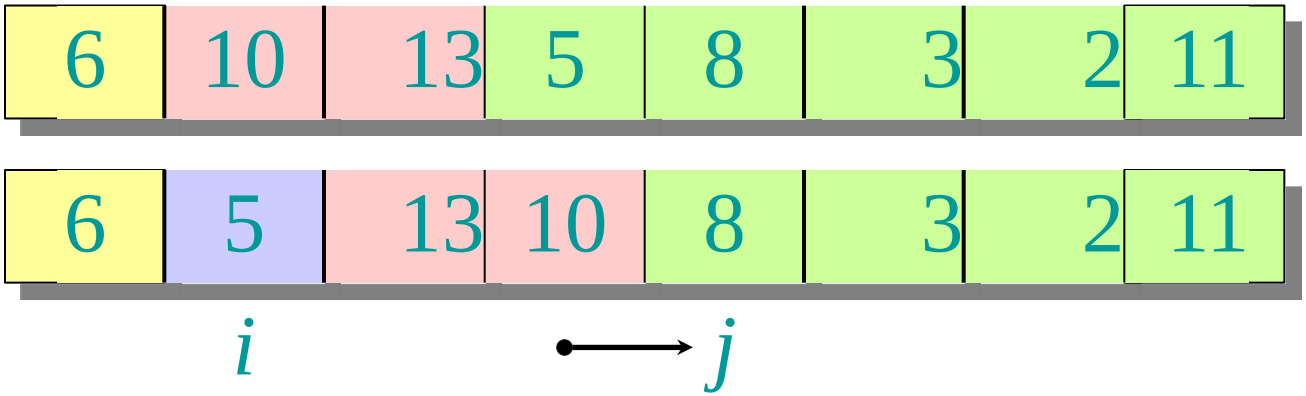


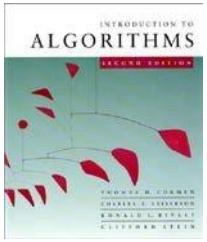
Example of partitioning



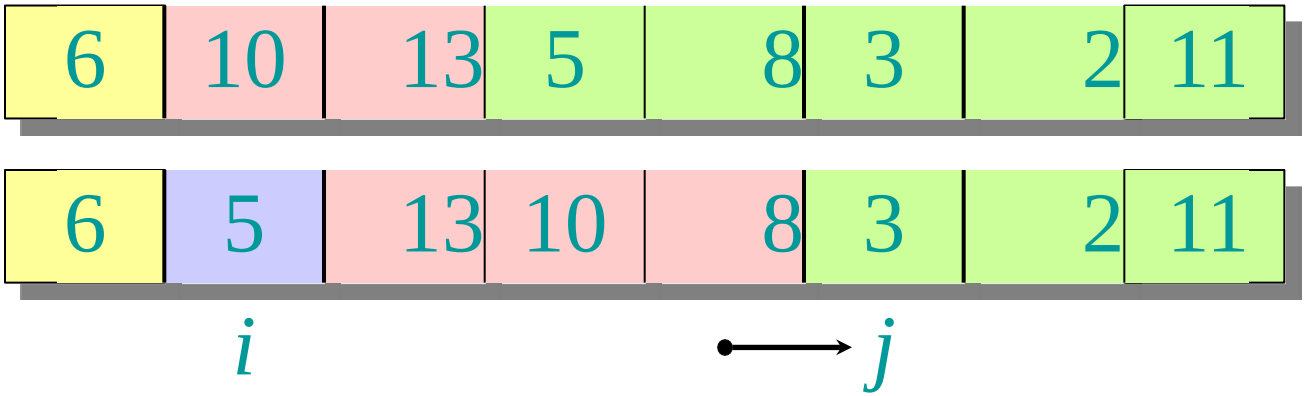


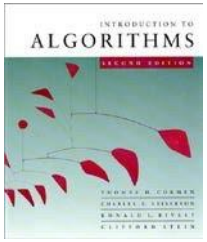
Example of partitioning



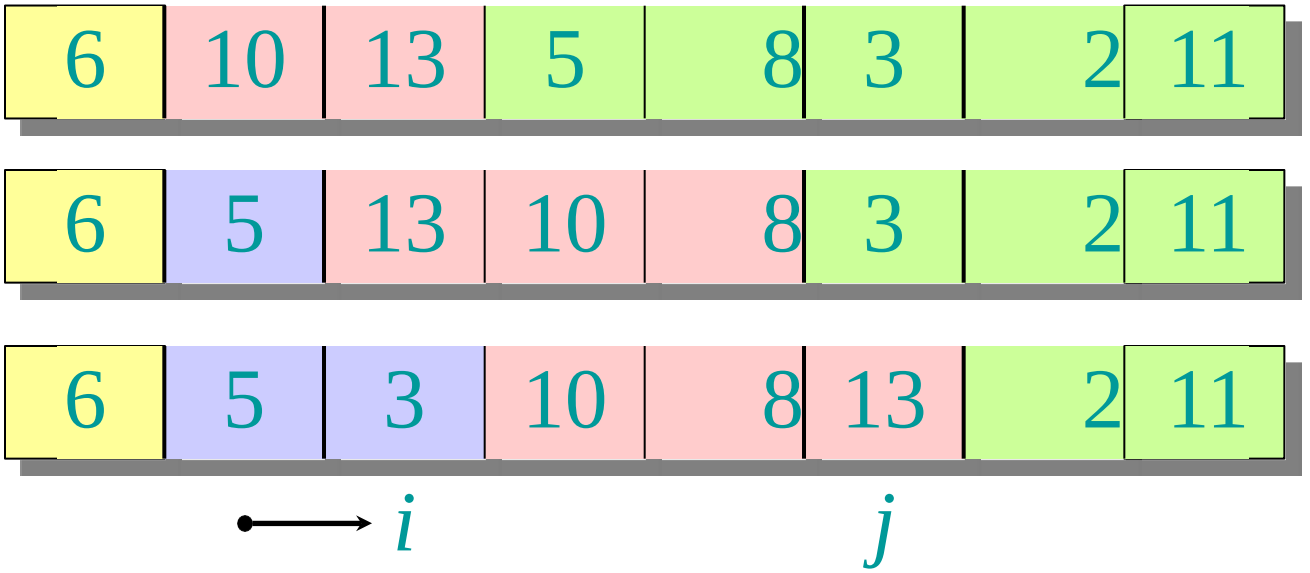


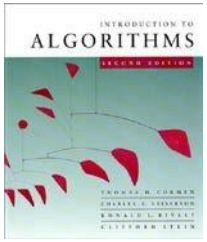
Example of partitioning



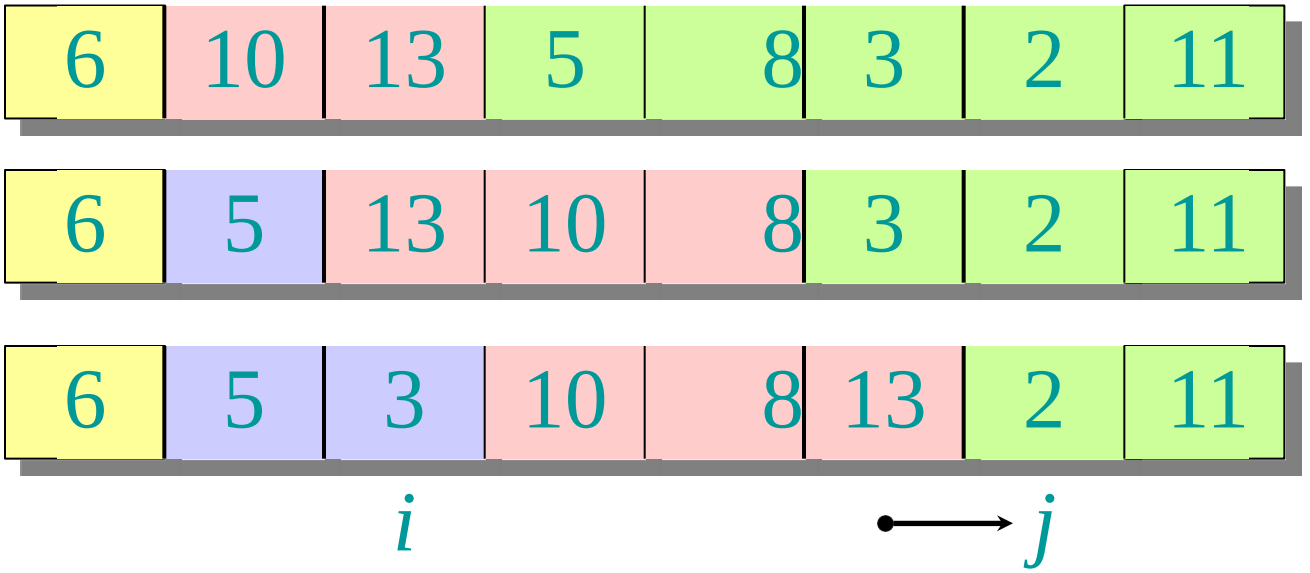


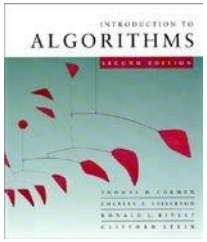
Example of partitioning



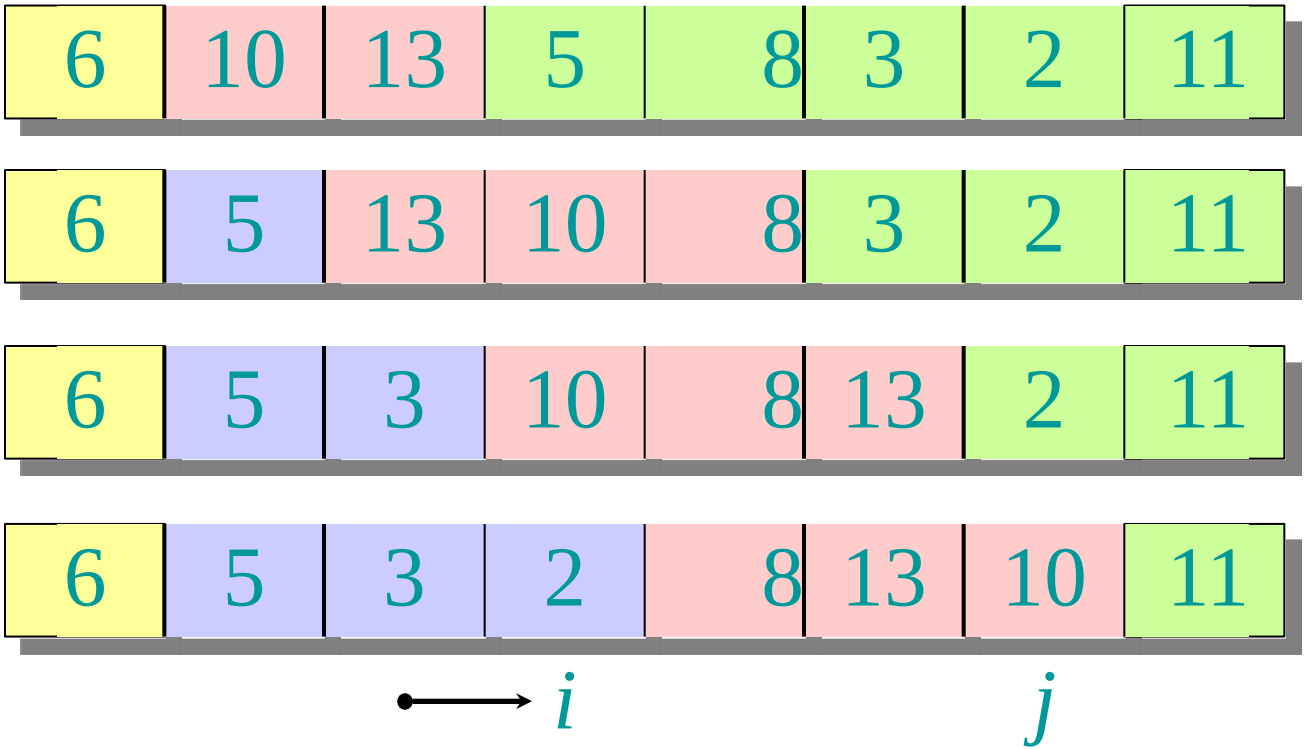


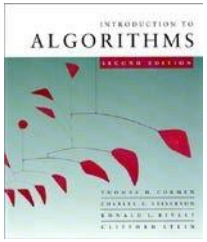
Example of partitioning



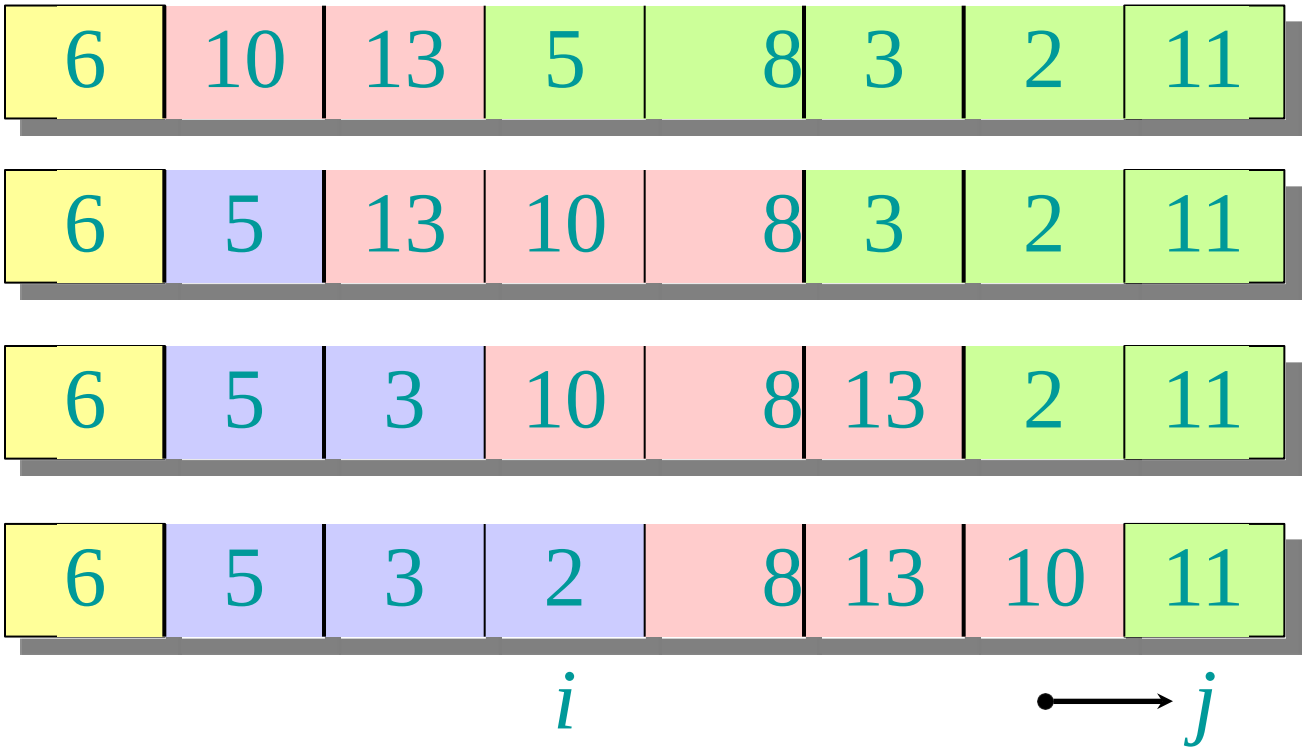


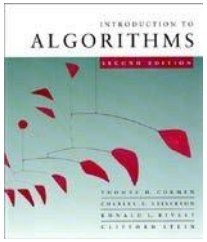
Example of partitioning



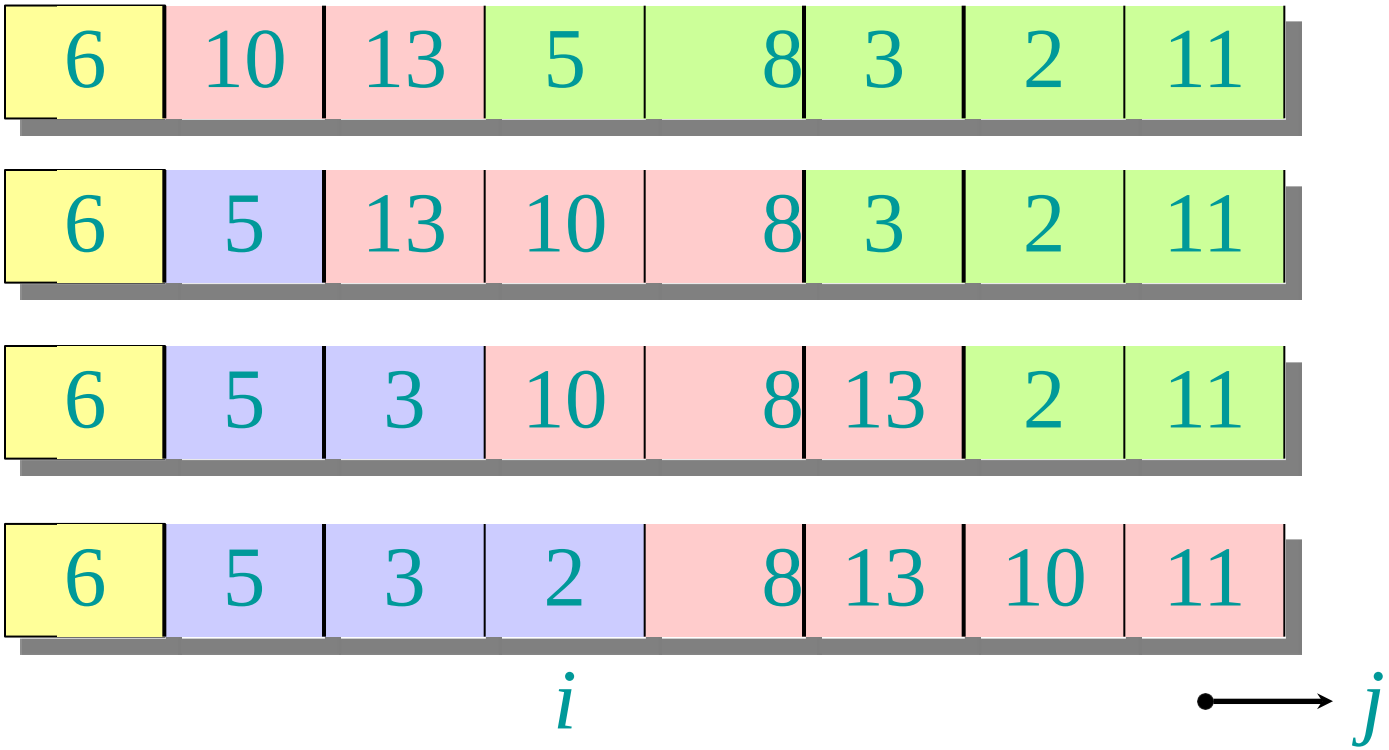


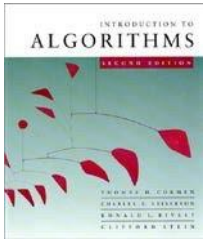
Example of partitioning





Example of partitioning

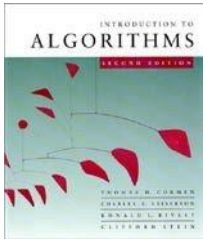




Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11

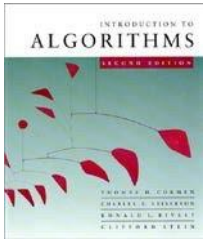
i



Pseudocode for quicksort

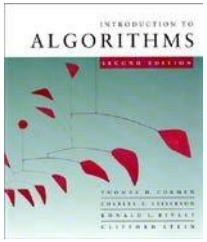
QUICKSORT(A, p, r)
if $p < r$
then $q \leftarrow$ PARTITION(A, p, r)
QUICKSORT(A, p, q–1)
QUICKSORT(A, q+1, r)

Initial call: QUICKSORT(A, 1, n)



Analysis of quicksort

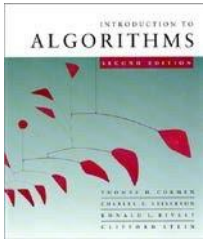
- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.



Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$T(n) = T(0) + T(n-1) + \Theta(n)$$
$$= \Theta(1) + T(n-1) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$
$$= \Theta(n^2)$$ ***(arithmetic series)***

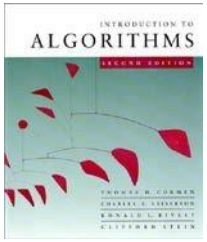


Worst-case recursion tree

$$T(n) = T(0) + T(n–1) + cn$$

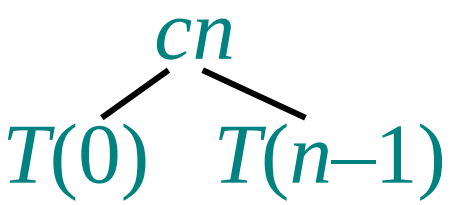
Worst-case recursion tree

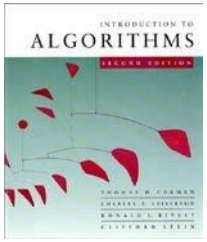
$T(n) = T(0) + T(n–1) + cn$ $T(n)$



Worst-case recursion tree

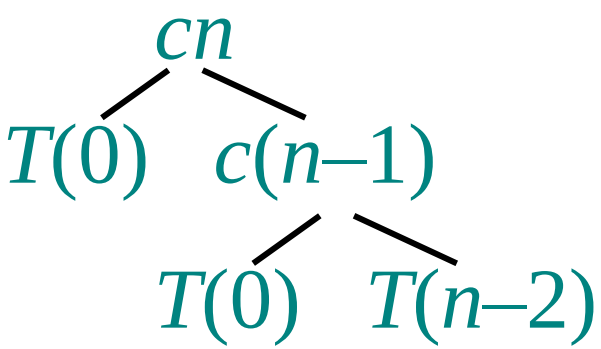
$$T(n) = T(0) + T(n–1) + cn$$

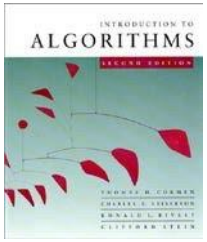




Worst-case recursion tree

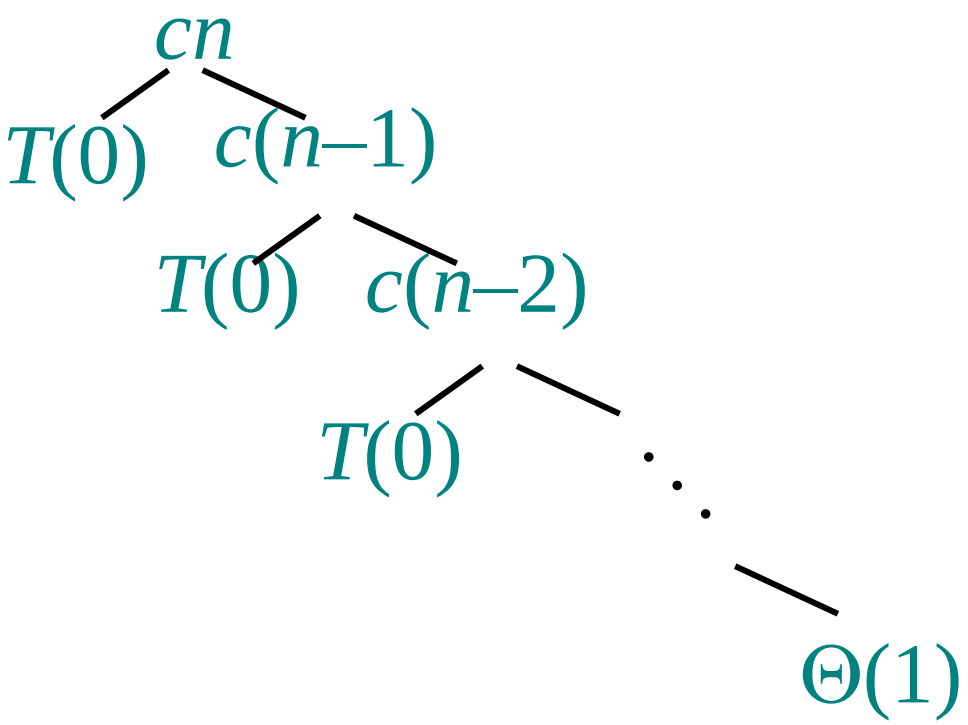
$$T(n) = T(0) + T(n–1) + cn$$

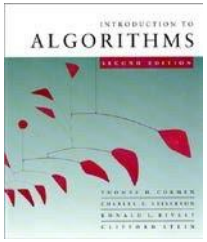




Worst-case recursion tree

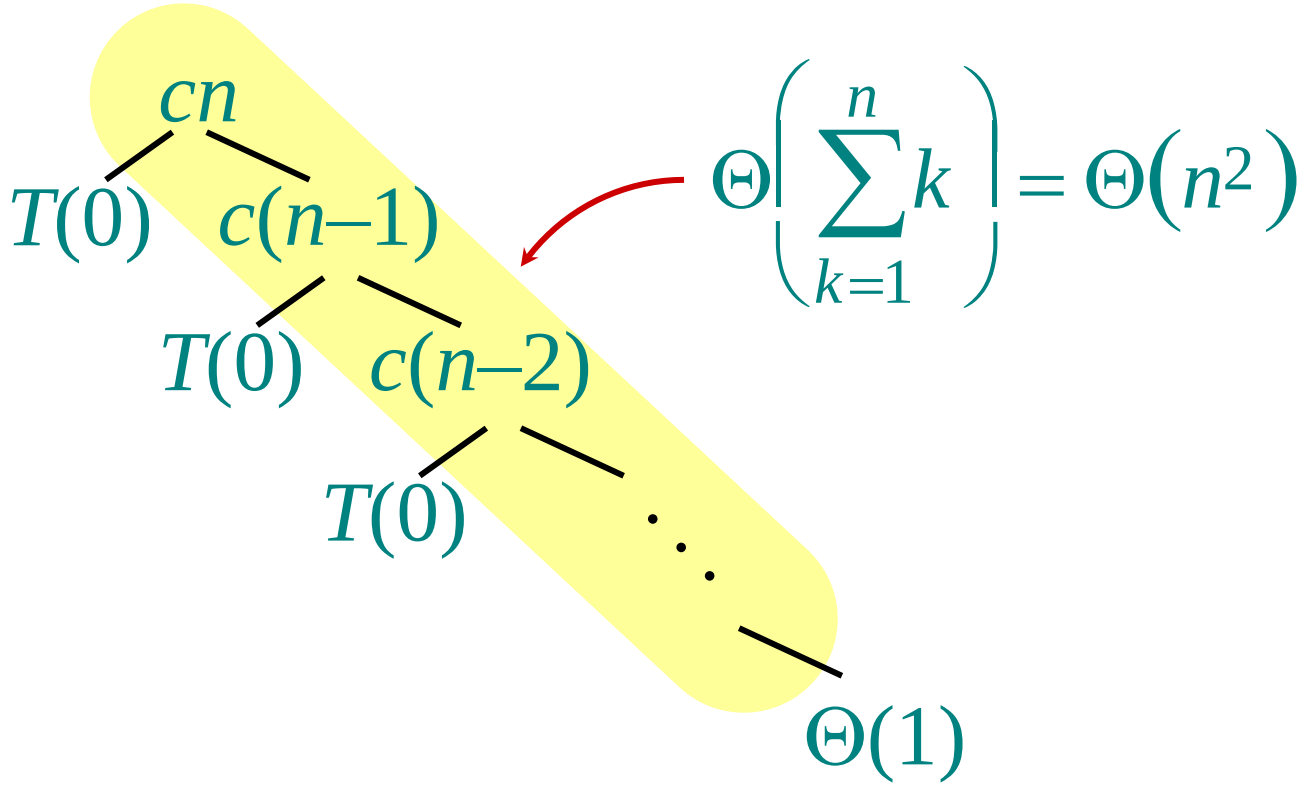
$$T(n) = T(0) + T(n–1) + cn$$

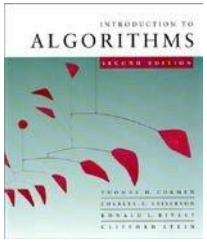




Worst-case recursion tree

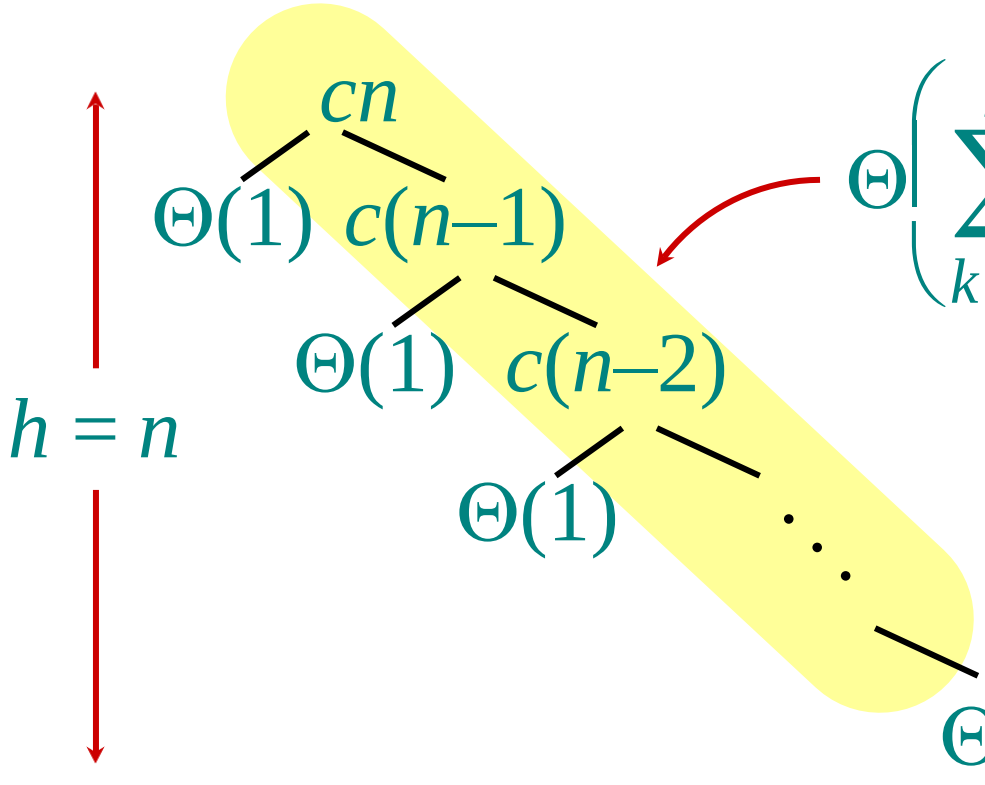
$$T(n) = T(0) + T(n-1) + cn$$





Worst-case recursion tree

$$T(n) = T(0) + T(n–1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

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

Worst Case Partitioning

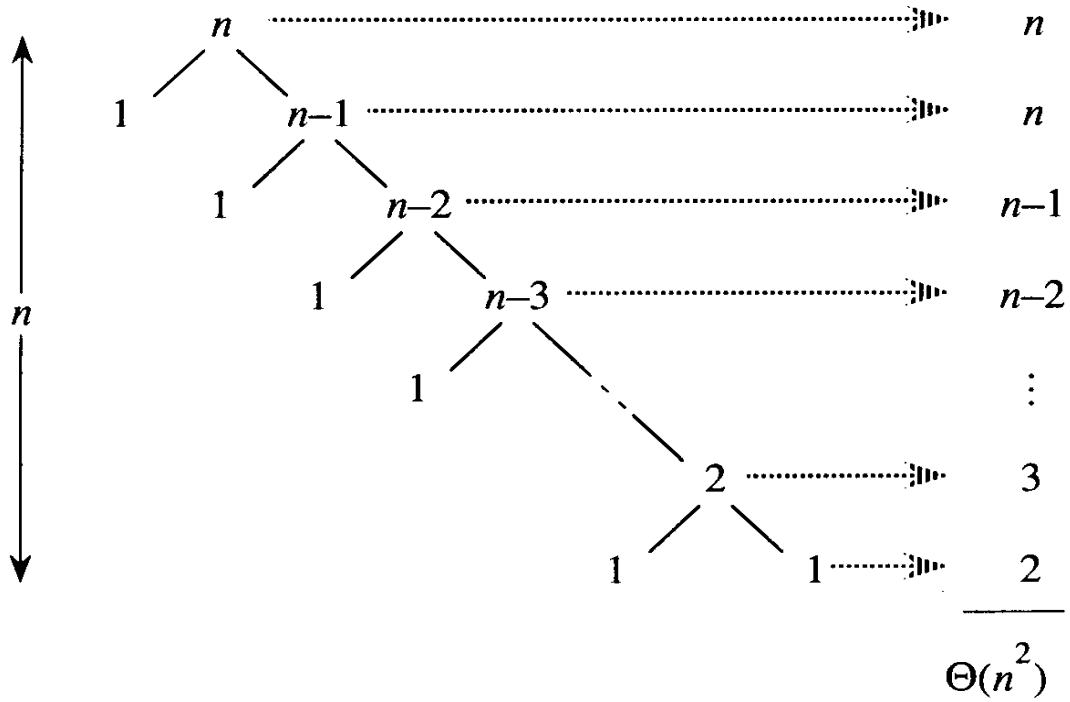


Figure 8.2 A recursion tree for QUICKSORT in which the PARTITION procedure always puts only a single element on one side of the partition (the worst case). The resulting running time is $\Theta(n^2)$.

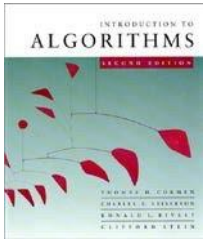
Worst Case Partitioning

- **Worst-Case Performance (unbalanced):**
  - $T(n) = T(0) + T(n-1) + \Theta(n)$
    - **partitioning takes $\Theta(n)$**

  $= \Theta(n^2)$

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = \Theta(n^2)$$

- **This occurs when**
  - **the input is completely sorted**
- **or when**
  - **the pivot is always the smallest (largest) element**



Best-case analysis

(For intuition only!)

If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n) \qquad \text{(same as merge sort)}$$

Best Case Partitioning

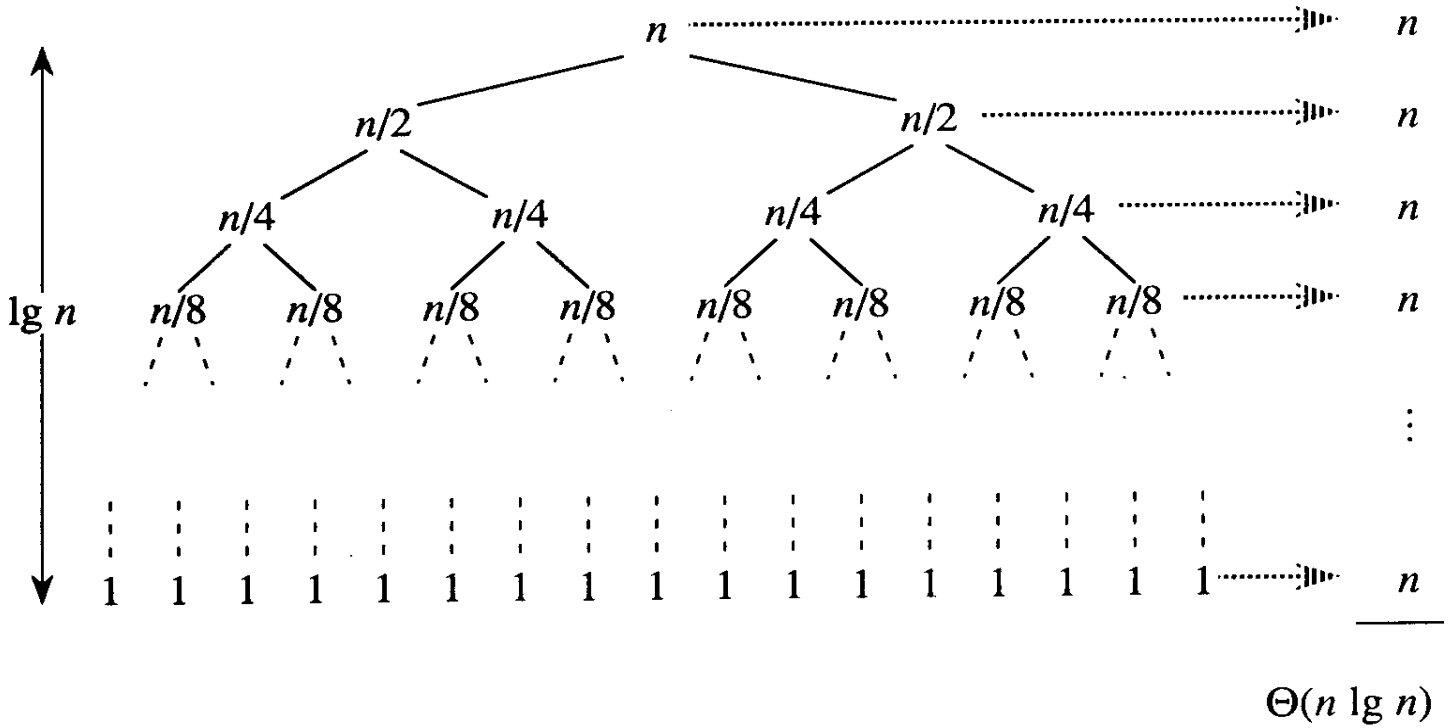
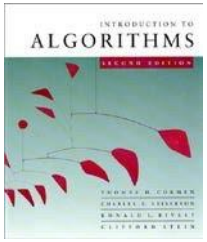


Figure 8.3 A recursion tree for QUICKSORT in which PARTITION always balances the two sides of the partition equally (the best case). The resulting running time is $\Theta(n \lg n)$.



Best-case analysis

(For intuition only!)

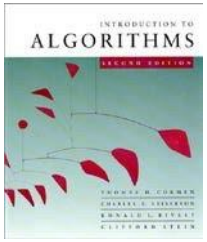
If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n) \qquad \text{(same as merge sort)}$$

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

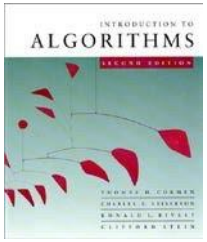
$$T(n) = T\left(\tfrac{1}{10} n\right) + T\left(\tfrac{9}{10} n\right) + \Theta(n)$$

What is the solution to this recurrence?

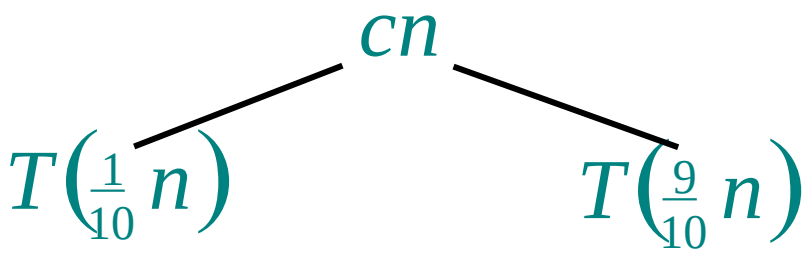


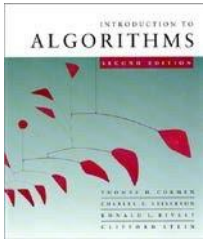
Analysis of “almost-best” case

$T(n)$

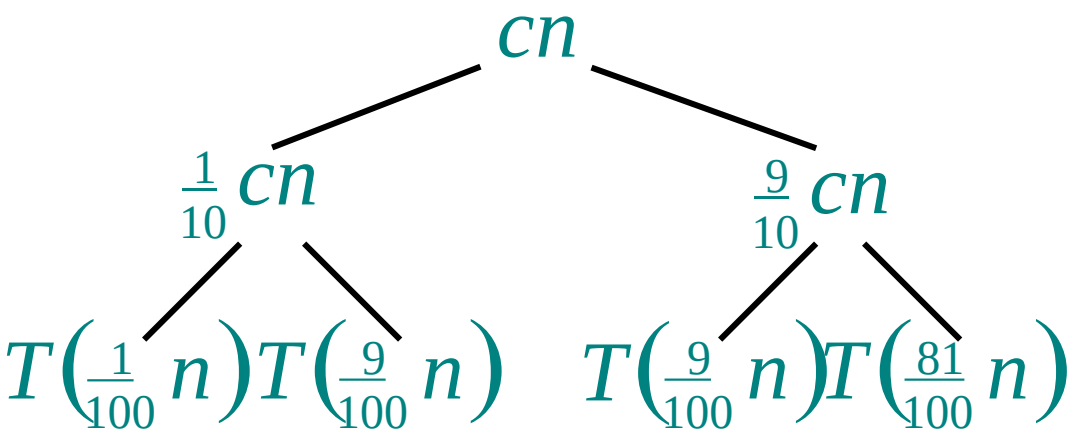


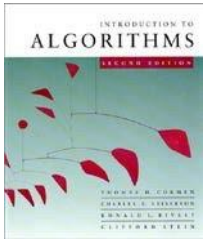
Analysis of "almost-best" case



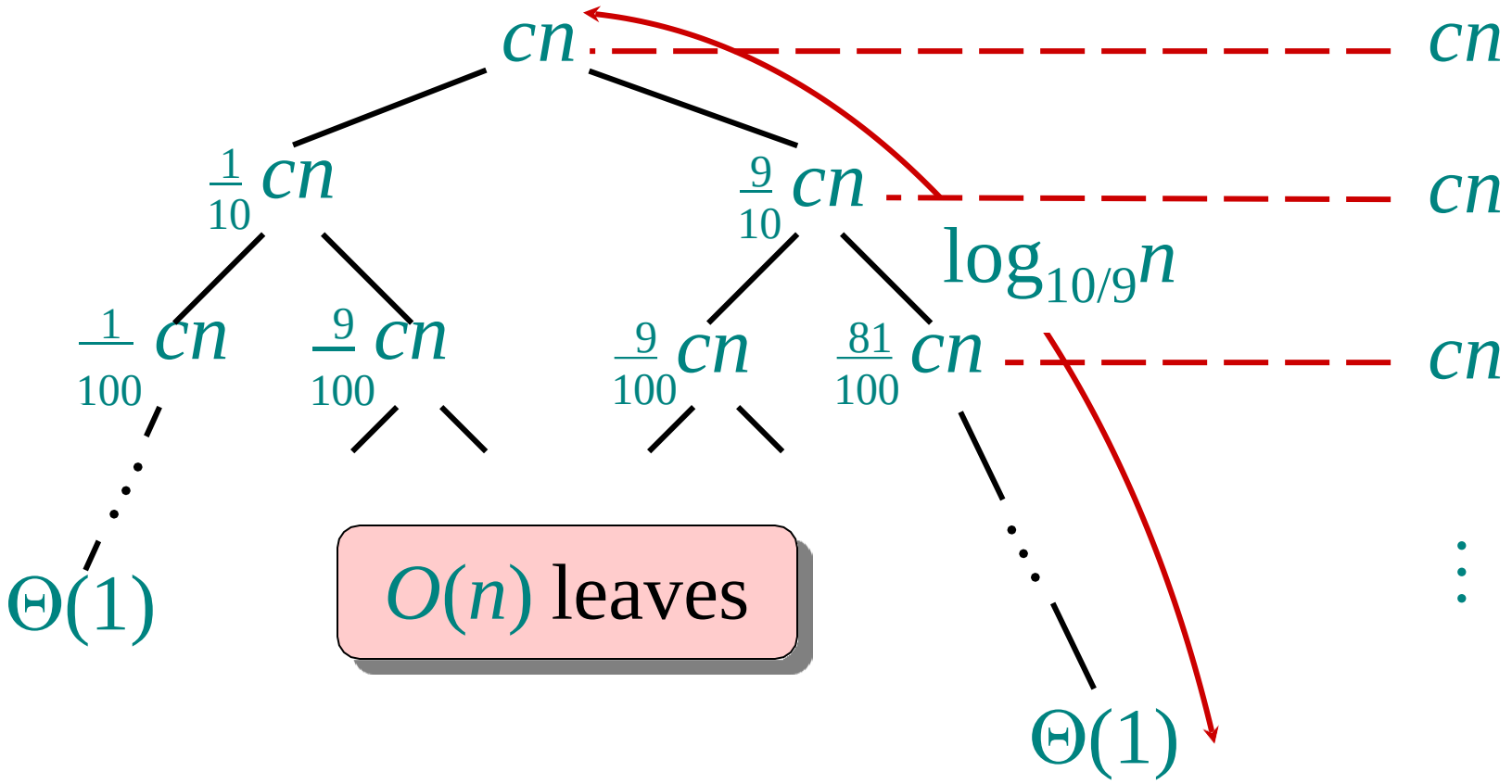


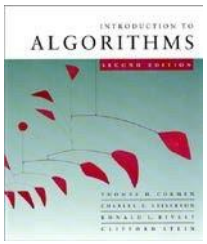
Analysis of "almost-best" case



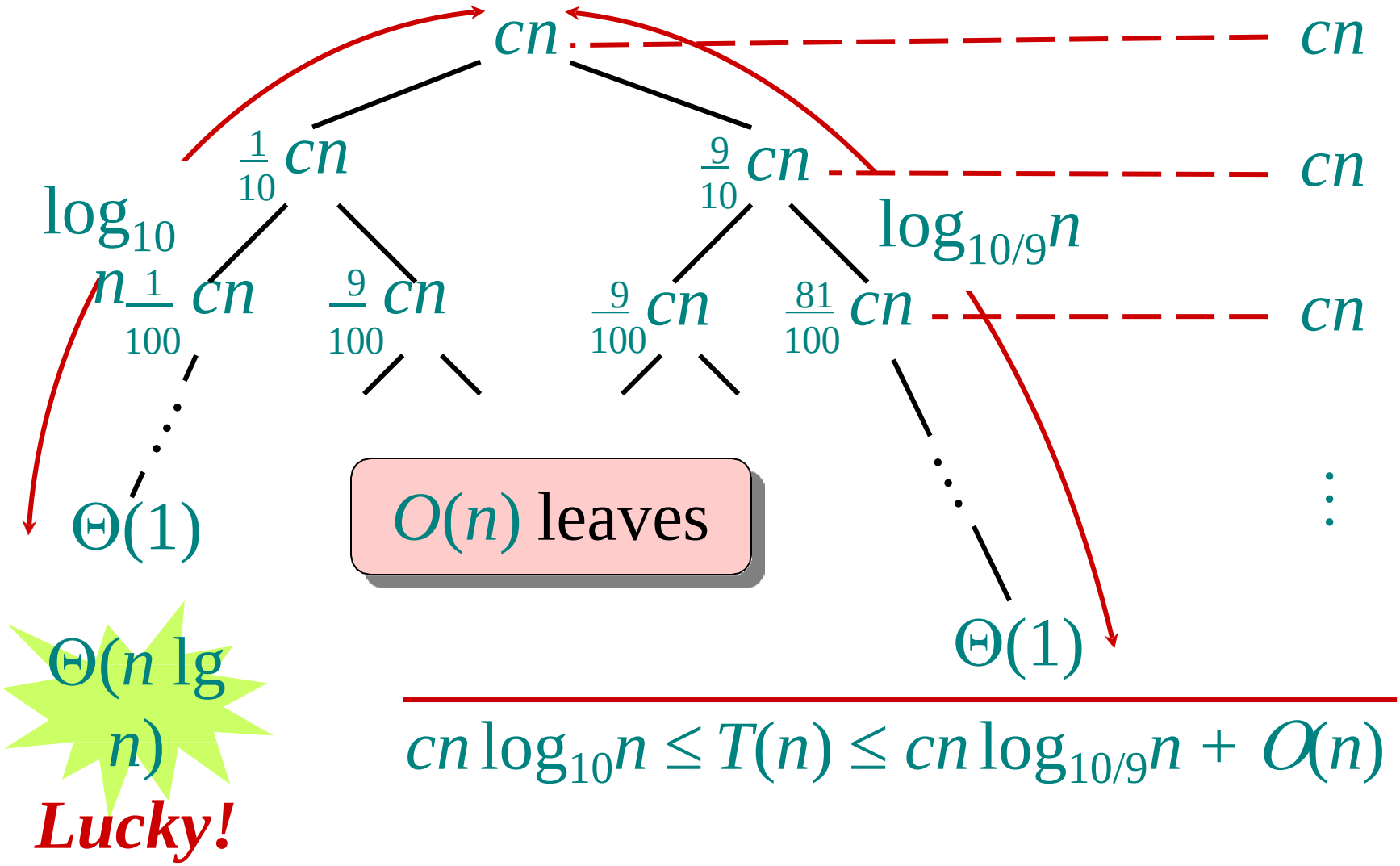


Analysis of "almost-best" case



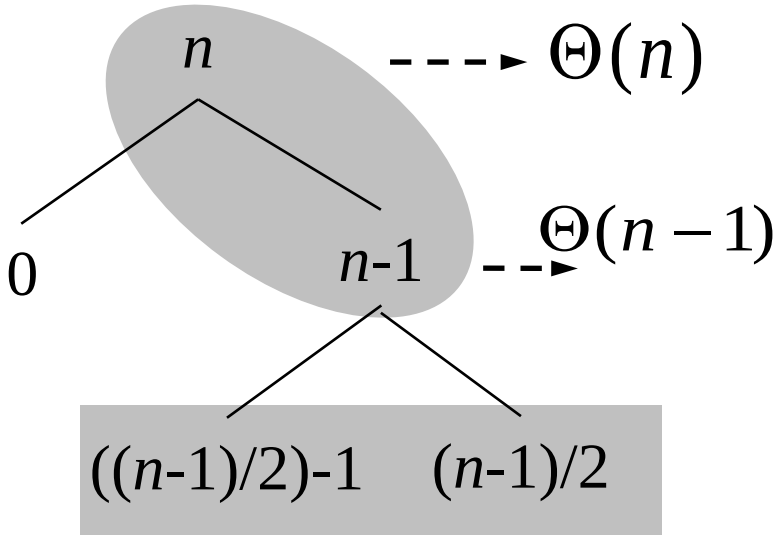


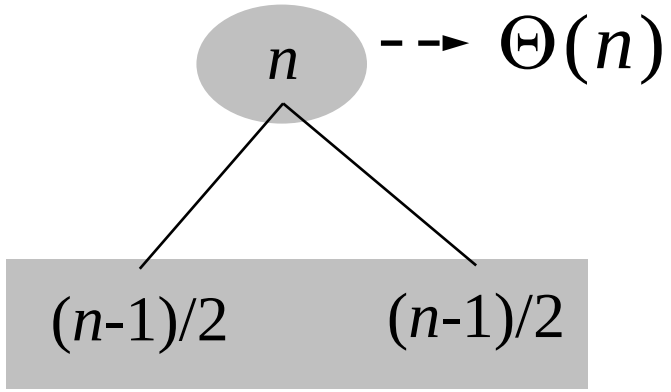
Analysis of "almost-best" case

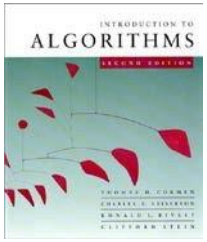


Average Case

- **What happens if we bad-split root node, then good-split the resulting size (*n*-1) node?**
  - **We end up with three subarrays, size**
    - **0, ((*n*-1)/2)-1, (*n*-1)/2**
  - **Combined cost of splits = Θ(*n*) + Θ(*n-1*) = Θ(*n*)**







More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

$L(n) = 2U(n/2) + \Theta(n)$ ***lucky***

$U(n) = L(n - 1) + \Theta(n)$ ***unlucky***

Solving:

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

Lucky!

How can we make sure we are usually lucky?

The combination of good and bad splits would result in $T(n) = O(n \lg n)$, but with slightly larger constant hidden by the O-notation

in-place algorithm

- **An in-place algorithm is an algorithm which transforms input using no auxiliary data structure.**
- **However a small amount of extra storage space is allowed for auxiliary variables.**
- **The input is usually overwritten by the output as the algorithm executes.**
- **In-place algorithm updates input sequence only through replacement or swapping of elements.**
- **An algorithm which is not in-place is sometimes called not-in-place or out-of-place**

Quicksort

- **Quicksort pros [advantage]:**
  - **Sorts in place**
  - **Sorts $O(n \lg n)$ in the average case**
  - **Very efficient in practice , it's quick**

- **Quicksort cons [disadvantage]:**
  - **Sorts $O(n^2)$ in the worst case**
  - **And the worst case doesn't happen often … sorted**

Randomized Quicksort

- **An algorithm is *randomized* if its behavior is determined not only by the input but also by values produced by a *random-number generator*.**

- **Exchange $A[r]$ with an element chosen at random from $A[p \ldots r]$ in Partition.**

- **This ensures that the pivot element is equally likely to be any of input elements.**

- **We can sometimes add randomization to an algorithm in order to obtain good average-case performance over all inputs.**

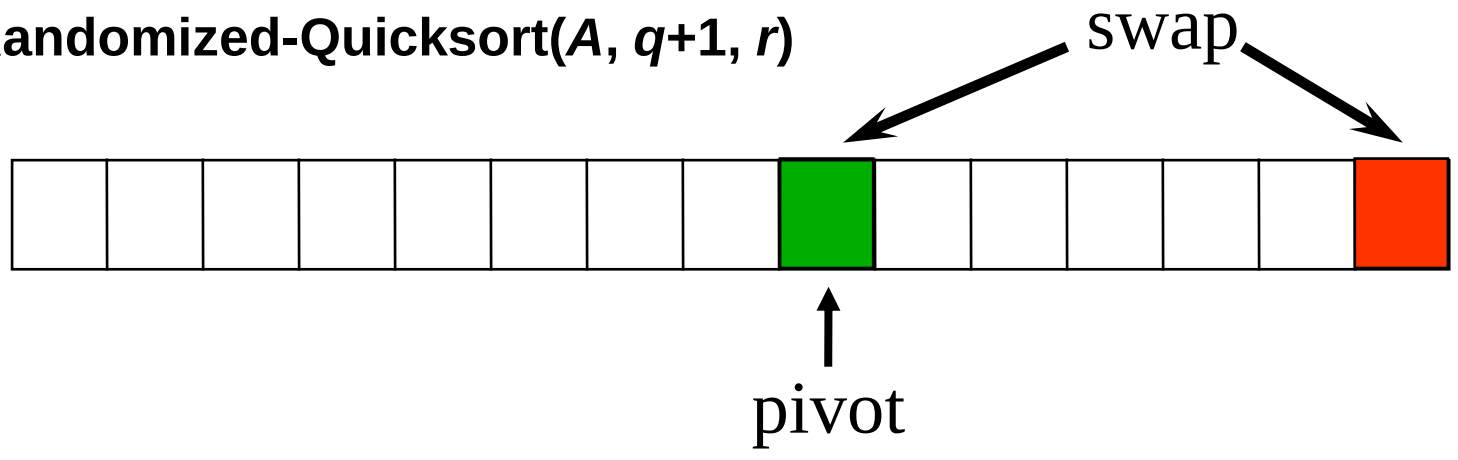
Randomized Quicksort

Randomized-Partition(*A, p, r*)

1. $i \leftarrow Random(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. return Partition(*A*, *p*, *r*)

Randomized-Quicksort(*A, p, r*)

1. if $p < r$
2. then $q \leftarrow$ Randomized-Partition(*A*, *p*, *r*)
3. Randomized-Quicksort(*A*, *p* , *q*-1)
4. Randomized-Quicksort(*A*, *q*+1, *r*)



Review: Analyzing Quicksort

- ***What will be the worst case for the algorithm?***
  - **Partition is always unbalanced**
- ***What will be the best case for the algorithm?***
  - **Partition is balanced**

Summary: Quicksort

- In worst-case, efficiency is $\Theta(n^2)$
  - But easy to avoid the worst-case
- On average, efficiency is $\Theta(n \lg n)$
- Better space-complexity than mergesort.
- In practice, runs fast and widely used

Practice Excercise

- **Select value at last index as pivot value and draw complete recurrence tree for quick sort. Describe its efficiency in terms of relating it with Best/worst/average case**

- 10,15,4,7,32,65,2,45,20
- 10,20,30,40,50,60,70,80,90
- 56,45,35,32,29,18,11,10,7,8
- 10,10,10,10,10,10,10,10,10,10

Practice Excercise

- **Select value at middle index of an array as pivot value and draw complete recurrence tree for quick sort. Describe its efficiency in terms of relating it with Best/worst/average case**

- 10,15,4,7,32,65,2,45,20
- 10,20,30,40,50,60,70,80,90
- 56,45,35,32,29,18,11,10,7,8
- 10,10,10,10,10,10,10,10,10,10

Practice Excercise

- **Select value at first index as pivot value and draw complete recurrence tree for quick sort. Describe its efficiency in terms of relating it with Best/worst/average case**

- 10,15,4,7,32,65,2,45,20
- 10,20,30,40,50,60,70,80,90
- 56,45,35,32,29,18,11,10,7,8
- 10,10,10,10,10,10,10,10,10,10

Practice Excercise

- **Select pivot value using randomized quick_sort and draw complete recurrence tree for quick sort. Describe its efficiency in terms of relating it with Best/worst/average case**

- 10,15,4,7,32,65,2,45,20
- 10,20,30,40,50,60,70,80,90
- 56,45,35,32,29,18,11,10,7,8
- 10,10,10,10,10,10,10,10,10,10

Reference

- **Introduction to Algorithms**
  - **Chapter # 7**
  - **Thomas H. Cormen**
  - **3rd Edition**