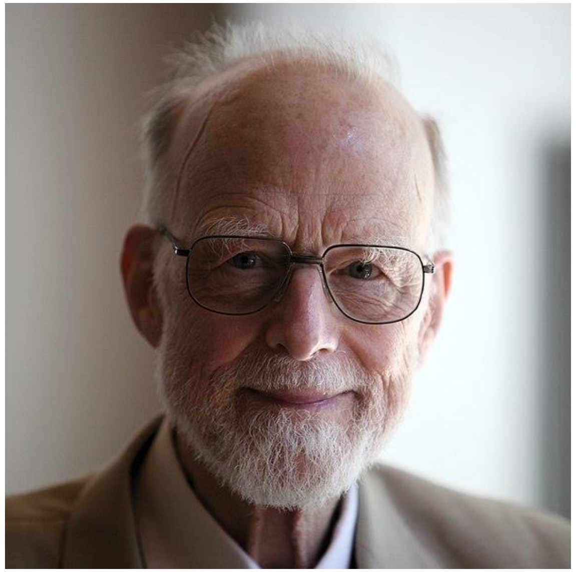
CSE3318: Divide & Conquer Algorithm
Quick Sort
by
Dr. Bhanu Jain

All slides are based on: ***Introduction to Algorithms***, by Thomas H. Cormen, Charles E. Leiserson, Ronald E. Rivest, Clifford Stein,3rd edition (CLRS)

Quick Sort

- Author: Tony Hoare, 1961
- Divide-and –conquer (and combine) algorithm
- In place sorting (like insertion sort, but not like merge sort)
- Very practical (with tuning)
- Has a worst-case running time of $\Theta(n^2)$
- Remarkably efficient on the average
- Expected running time is $\Theta(n\lg n)$

Tony Hoare in 2011



Charles Antony Richard Hoare

Born	11 January 1934 (age 91) Colombo, British Ceylon
Residence	Cambridge
Other names	C. A. R. Hoare
Alma mater	•University of Oxford (BA) •Moscow State University

Credit: https://en.wikipedia.org/wiki/Tony_Hoare

Divide and Conquer Algorithms

- Division of problem yields a subproblems, each of which is *1*/b the size of the original
- D(*n*) time to divide the problem into subproblems
- C(n) time to combine the solutions
- *c* is some constant

$$T(n) = \begin{cases} \Theta(1) \text{ if } n \le c; \\ aT(n/b) + D(n) + C(n) \text{ otherwise.} \end{cases}$$

Divide and Conquer Algorithms: Quick Sort

Quicksort an n-element array A[p..r]:

1. **Divide:** Partition the array A[p..r] into two 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 less than or equal to each element of A[q+1..r] Compute the index q as part of this partitioning procedure
2. **Conquer:** Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort
3. **Combine:** No additional work is required as the entire array A[p..r] is now already sorted

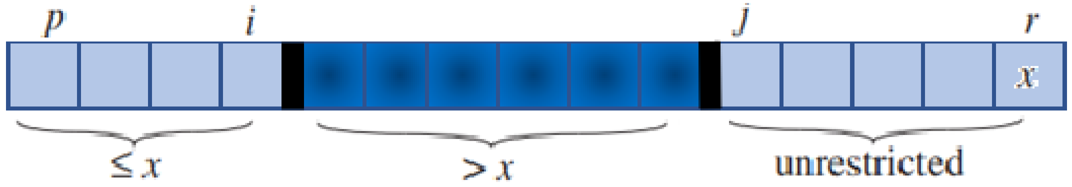
Quicksort (A, p, r)

```
If r>p
    q= Partition (A, p, r)
    Quicksort (A, p, q-1)
    Quicksort (A, q+1, r)
```

Image Credit : Courtesy Dr. Junzhou Huang

Partitioning the array

The key to the algorithm is the **PARTITION procedure**, which rearranges the subarray A[p..r] in place.



- PARTITION always selects an element ***x* = A[r]** as a pivot
- Pivot element is used to partition the subarray **A[p..r]**
- Procedure partitions the array into 4 (possibly empty) regions
  1. The values in **A[p..i]** are all less than or equal to x.
  2. The values in **A[i+1..j-1]** are all greater than x
  3. The subarray **A[j..r-1]** can take on any values.
  4. A[r]=x

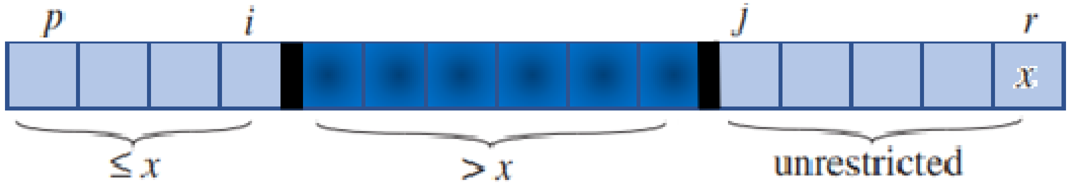
a) If p =< k ≤ i, then A[k] ≤ x

b) If i+1 =< k ≤ j – 1, then A[k] > x

c) If k=r, then A[k] = x

Partitioning the array

The key to the algorithm is the **PARTITION** procedure, which rearranges the subarray A[p..r] in place.



- PARTITION always selects an element *x* = A[r] as a pivot
- Pivot element is used to partition the subarray A[p..r]
- Procedure partitions the array into 4 (possibly empty) regions
- At the start of each iteration of the for loop in lines 3–6, t for any array index ***k***

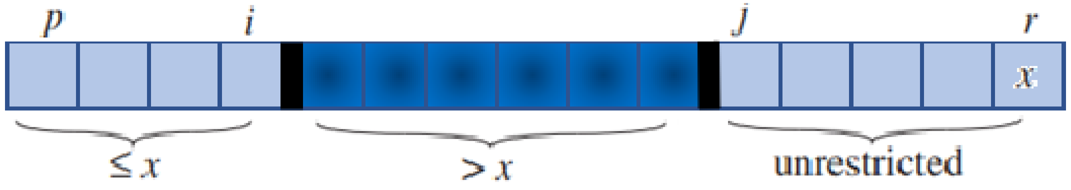
Partition (A, p, r)

```
1   x: = A[r]
2   i = p – 1
3   for j = p to r – 1
4         if A[j] ≤x
5               i = i + 1
6               swap A[i] and A[j]
7   swap A[i+1] and A[r]
8   return i + 1
```

a) If p =< k ≤ i, then A[k] ≤ x
b) If i+1 =< k ≤ j – 1, then A[k] > x
c) If k=r, then A[k] = x

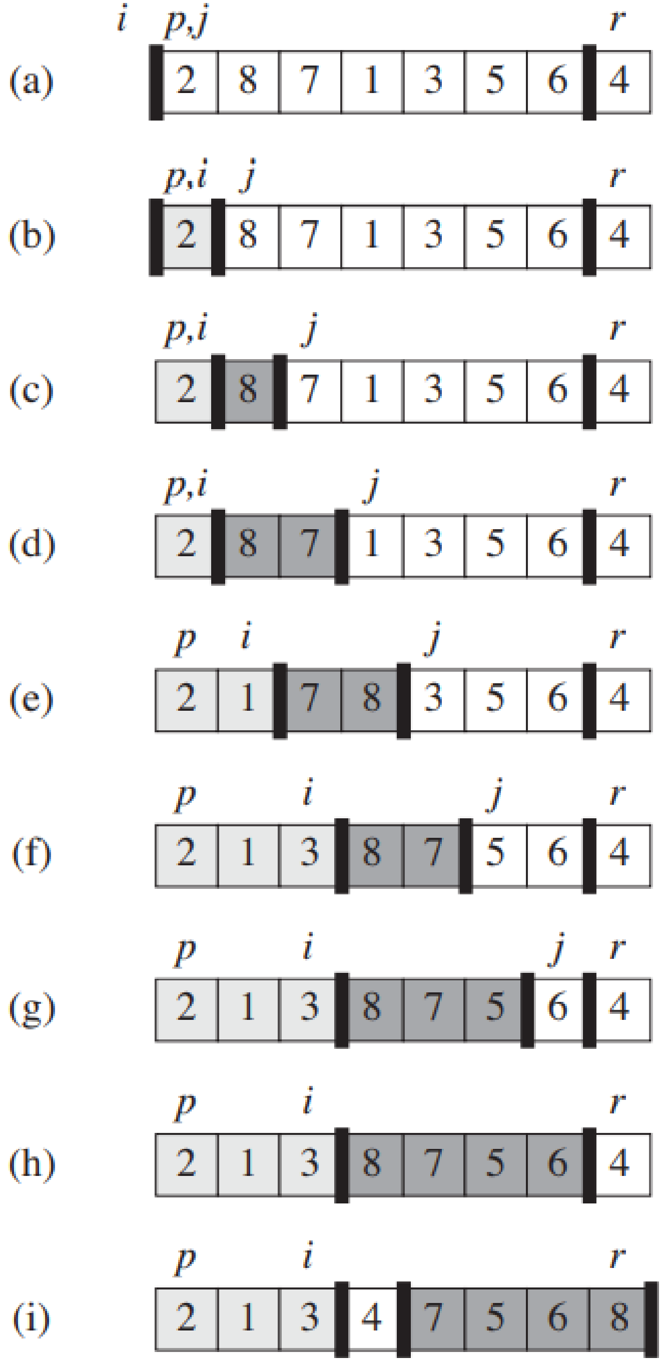
Partitioning the array

The key to the algorithm is the **PARTITION procedure**, which rearranges the subarray A[p..r] in place.



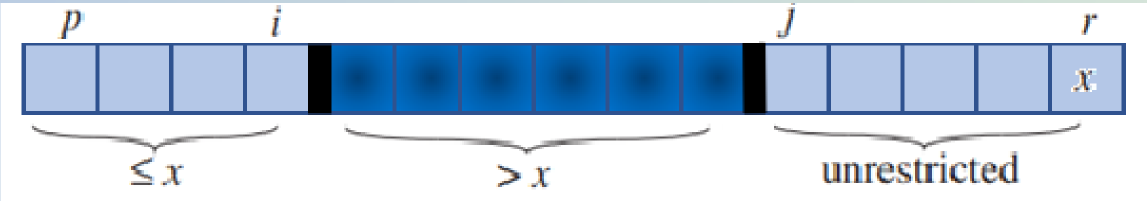
Partition (A, p, r)

```
x: = A[r]
i = p – 1
for j = p to r – 1
      if A[j] ≤x
            i = i + 1
            swap A[i] and A[j]
swap A[i+1] and A[r]
return i + 1
```



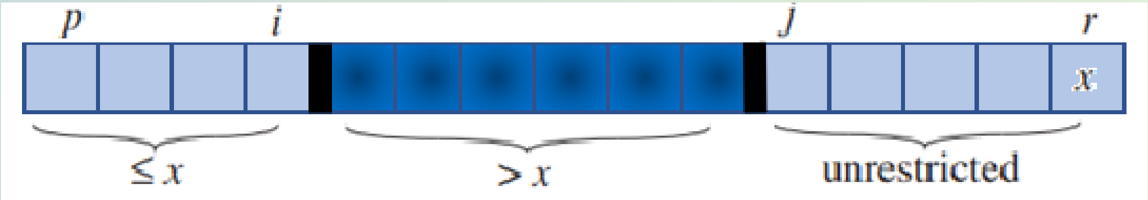
Performance of Quick Sort

- The running time of quicksort depends on whether the partitioning is balanced or unbalanced
- Balanced/Unbalanced partition depends on the pivot.
- If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort
- If the partitioning is unbalanced, the algorithm runs asymptotically as slow as insertion sort



Partition (A, p, r)

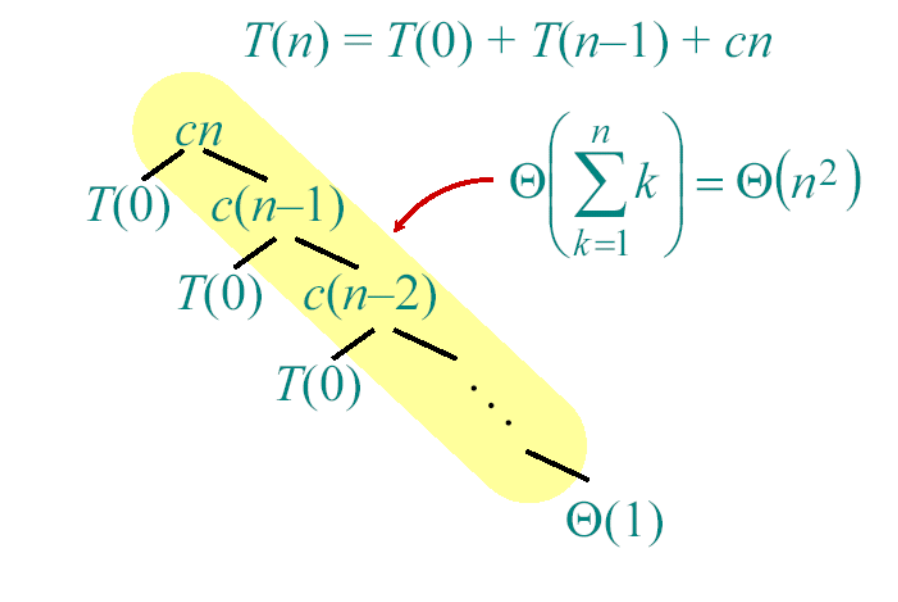
```
x: = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤x
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i + 1
```



Analysis of Quick Sort: Worst-case

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input element s may exist.
- Let T(n) = worst-case running time on array of n elements
- If the partitioning is unbalanced, the algorithm runs asymptotically as slow as insertion sort
- Input sorted or reverse sorted
- Partition around min or max element. Partitioning cost $\Theta(n)$
- One side of partition always has no elements.
- One subproblem with n -1 elements and one with 0 elements

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2) \qquad \textit{\textbf{(arithmetic series)}}
\end{aligned}
$$



Quick Sort: Best-case Analysis

(For intuition only!)

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

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

(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?

Quick Sort: Almost Best-case Analysis

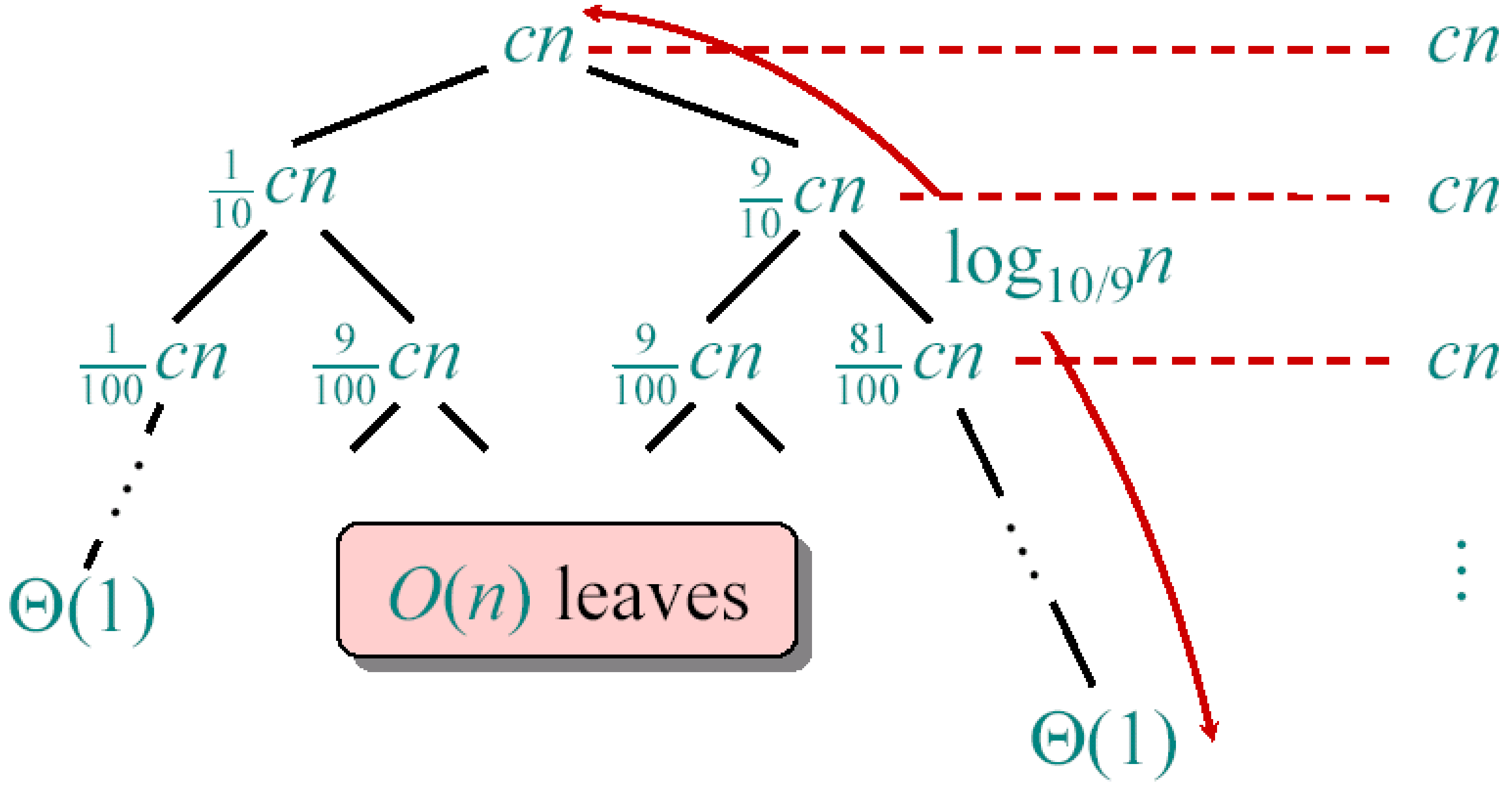


Image Credit : Courtesy Dr. Junzhou Huang

Quick Sort: Almost Best-case Analysis

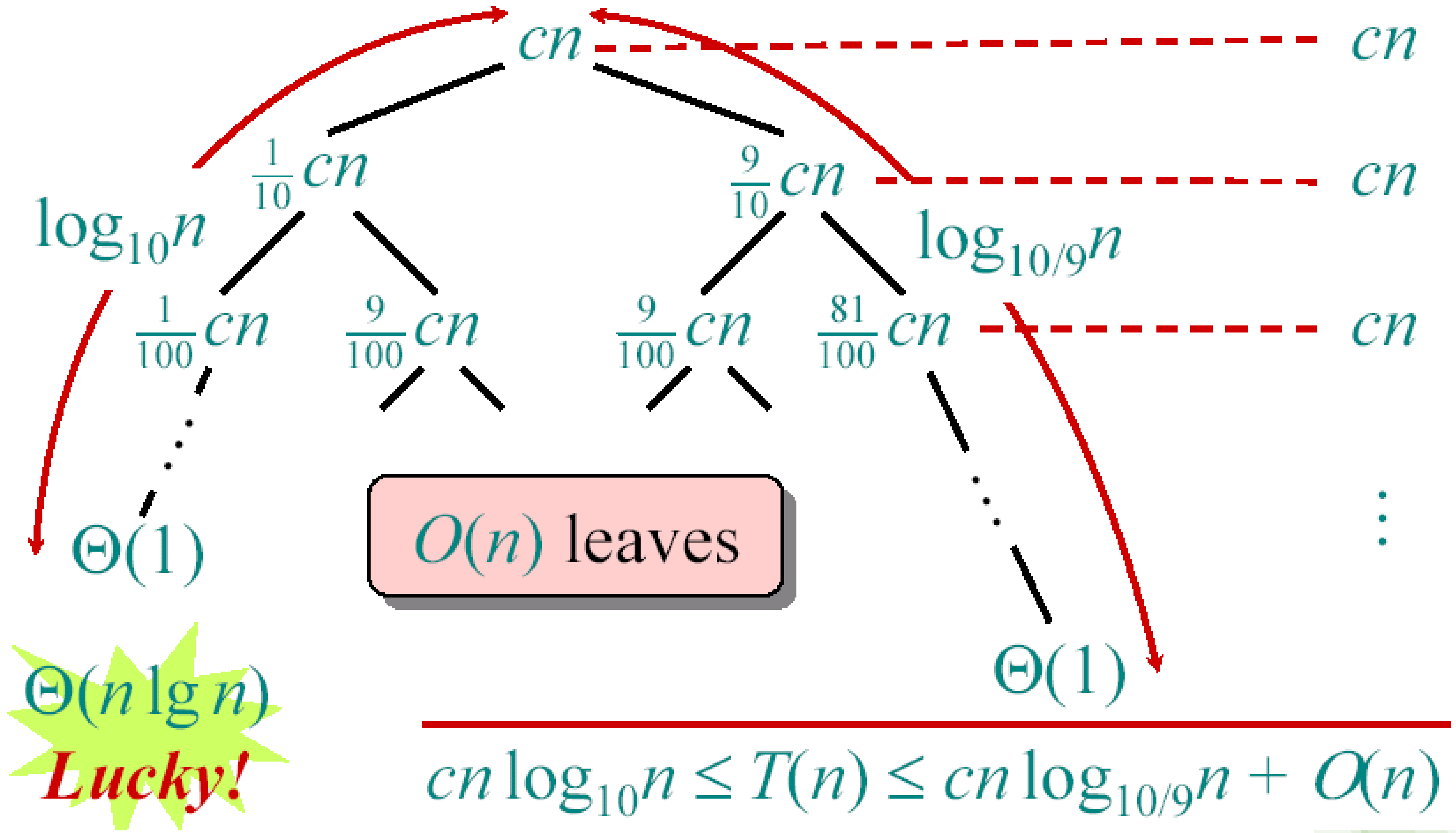


Image Credit : Courtesy Dr. Junzhou Huang

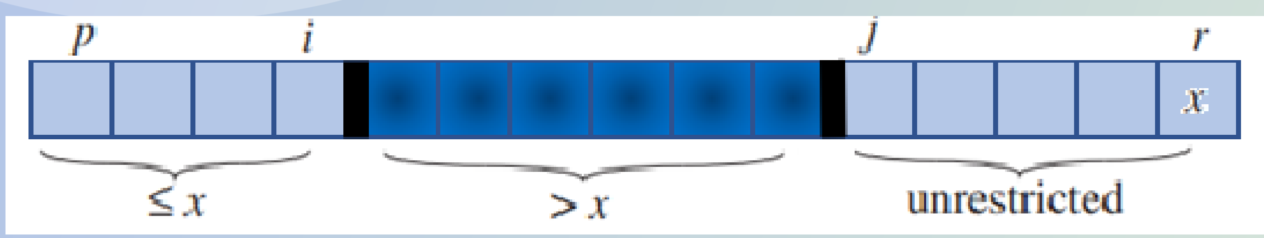
Randomized Quick Sort

- **Important Features**
- **Randomized Pivot Selection**: A random element is chosen as the pivot for partitioning, making the process unpredictable.
- **Input-Order Independence**: The algorithm's performance does not rely on how the input data is arranged.
- **Robust Against Input Distribution**: No prior knowledge or assumptions about the input's distribution are required.
- **Mitigates Predictable Worst-Case Scenarios**: The worst-case scenario arises solely from the random pivot choices, not from specific input patterns.

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$(A, 1, n)$

Randomized Quick Sort




```
QUICKSORT(A, p, r)
  if p < r
    then q ← PARTITION(A, p, r)
      QUICKSORT(A, p, q–1)
      QUICKSORT(A, q+1, r)
```

Initial call: QUICKSORT(A, 1, n)

Randomized-Partition (A, p, r)

```
If r>p
    i:= RANDOM (p, r)
    Swap (A[r], A[i])
    Return PARTITION (A, p, r)
```

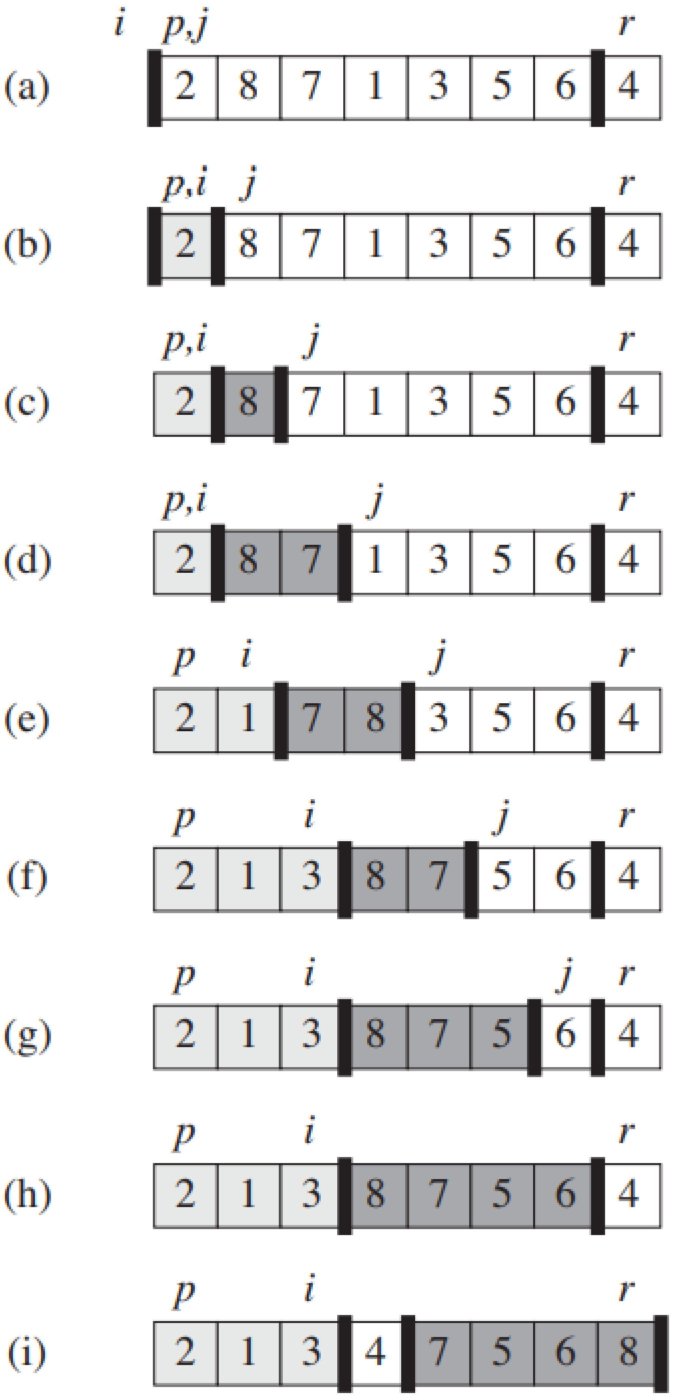
Partition (A, p, r)

```
x: = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤x
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i + 1
```

Randomized-Quicksort (A, p, r)

```
If r>p
    q= RANDOMIZED-PARTITION (A, p, r)
    Randomized-Quicksort (A, p, q-1)
    Randomized-Quicksort (A, q+1, r)
```

Partitioning the array



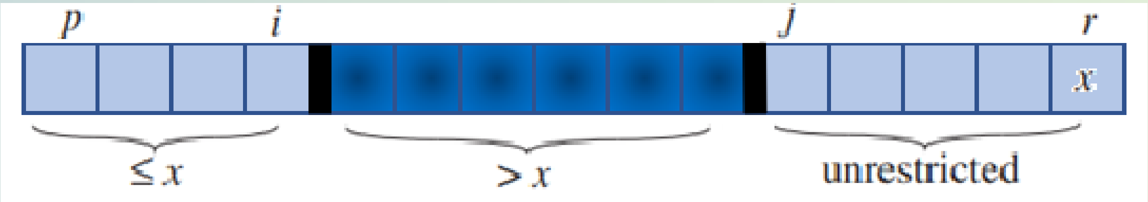
p, i, j, arr
1 0 1 [2, 8, 7, 1, 3, 5, 6, 4]
1 1 2 [2, 8, 7, 1, 3, 5, 6, 4]
1 1 3 [2, 8, 7, 1, 3, 5, 6, 4]
1 1 4 [2, 8, 7, 1, 3, 5, 6, 4]
1 2 5 [2, 1, 7, 8, 3, 5, 6, 4]
1 3 6 [2, 1, 3, 8, 7, 5, 6, 4]
1 3 7 [2, 1, 3, 8, 7, 5, 6, 4]
1 3 _ [2, 1, 3, 8, 7, 5, 6, 4]
1 3 _ [2, 1, 3, 4, 7, 5, 6, 8]

Partition (A, p, r)

```
x: = A[r]
i = p – 1
for j = p to r – 1
    if A[j] ≤x
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i + 1
```

Randomized Quick Sort

Quick Sort Summary



Why It Excels

1. Versatile and Reliable :

Works well with all kinds of data and input sizes, making it a dependable sorting choice.

2. Faster in Practice

Often beats merge sort due to its in-place sorting, avoiding extra memory usage and overhead.

3. Easily Tuned for Speed

Simple tweaks like optimized pivot selection or handling small arrays with insertion sort can make it even faster.

4. Cache and Memory Friendly

Quicksort uses memory efficiently by accessing data locally, reducing cache misses and working smoothly with virtual memory systems.

The END!