

# Design and Analysis of Algorithms

## L12: Quicksort

Dr. Ram P Rustagi  
Sem IV (2019-H1)  
Dept of CSE, KSIT/KSSEM  
[rprustagi@ksit.edu.in](mailto:rprustagi@ksit.edu.in)

# Resources

- Text book 1: Levitin (QuickSort)
-

# Sort Algorithms

- Bubble sort
- Selection sort
- Insertion sort
- Mergesort
- **Quicksort**
- Shell sort
- Heap sort
- Radix sort

# Quick Sort

- Introduced by Hoare in 1960
- MergeSort shortcomings
  - No inplace sort
  - Extra space could be costly
  - Inherently recursive
    - recursive calls and returns are expensive
- Purpose of quicksort
  - Overcome shortcomings of mergesort
  - Extra space is caused by Merge operation
    - Can we avoid merge operation
    - Merging happens because elements in left half move to right half and vice versa
  - Can we divide such elements in left half are always less than elements in right half?
    - No need to merge (no extra space required)

# Quick Sort

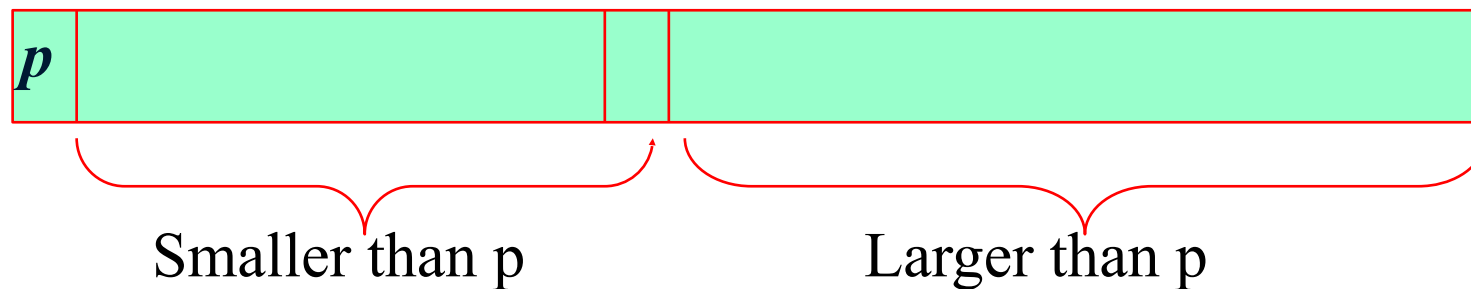
- What results in merging?
  - Some elements in right side need to move to left side
    - These are smaller than some elements in left side
- Objective
  - Divide in such a way that elements in left are always smaller than elements in right
  - Can we use divide and conquer?
  - Can we find middle value (median) and put in center?
- Method
  - Assume we have median  $m$  and placed in the middle
  - Move everything less than it to the left
  - Move everything greater than it to the right
- Claim: we can move everything in linear time.
  - Use the process recursively (divide and conquer)

# Quick Sort

- Division into parts,
  - Each part of size  $n/2$ ,
  - $n$  operations (moving to left and right of median)
  - Time complexity same as that of mergesort, i.e.
    - $T(n) = 2T(n/2) + \Theta(n)$   
 $= \Theta(n \log_2 n)$
- Challenge: how to find the median?
  - Finding median requires sorting of elements
    - Which is what we want to achieve
    - A classic chicken and egg problem?
- How to approach?
  - Pick some value (not necessarily the median)
  - Follow the steps as if it is median (pivot)

# QuickSort

- A highly efficient algorithm
- Pick a pivot, divide input array in smaller arrays using the pivot (a specified value)
  - One array contains smaller values
  - Other array contains larger values
- Exchange the pivot with last element in first array
  - pivot is in its final position
- Sort the sub arrays recursively

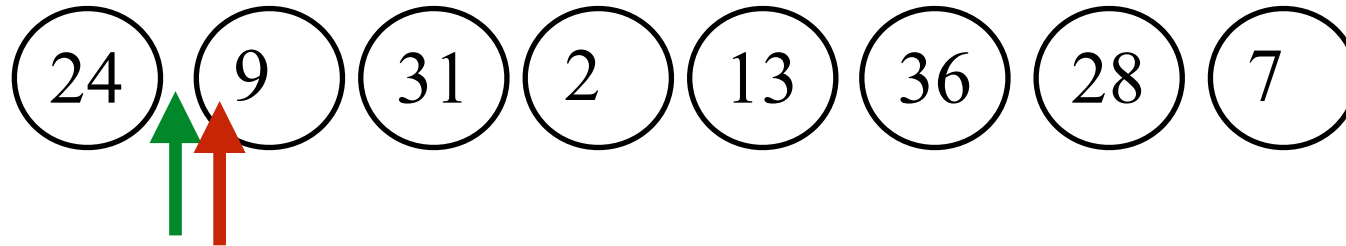


# QuickSort Algorithm: Steps

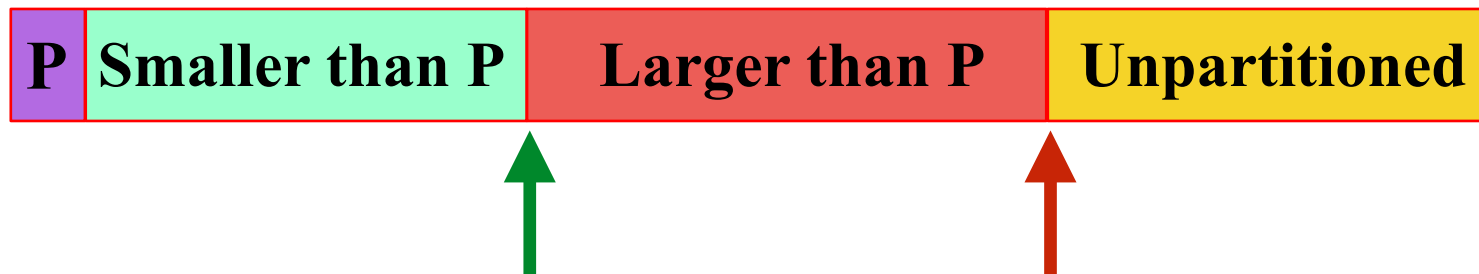
- Select a *pivot* (partitioning element)
  - e.g. 1<sup>st</sup> element or last element.
  - You can choose any element and swap it with last element
- Rearrange the array as follows i.e move pivot between lower and upper partition
  - All elements in first  $s-1$  positions are  $\leq$  pivot
  - All elements in remaining  $n-s$  positions  $\geq$  pivot
- Repeat the process



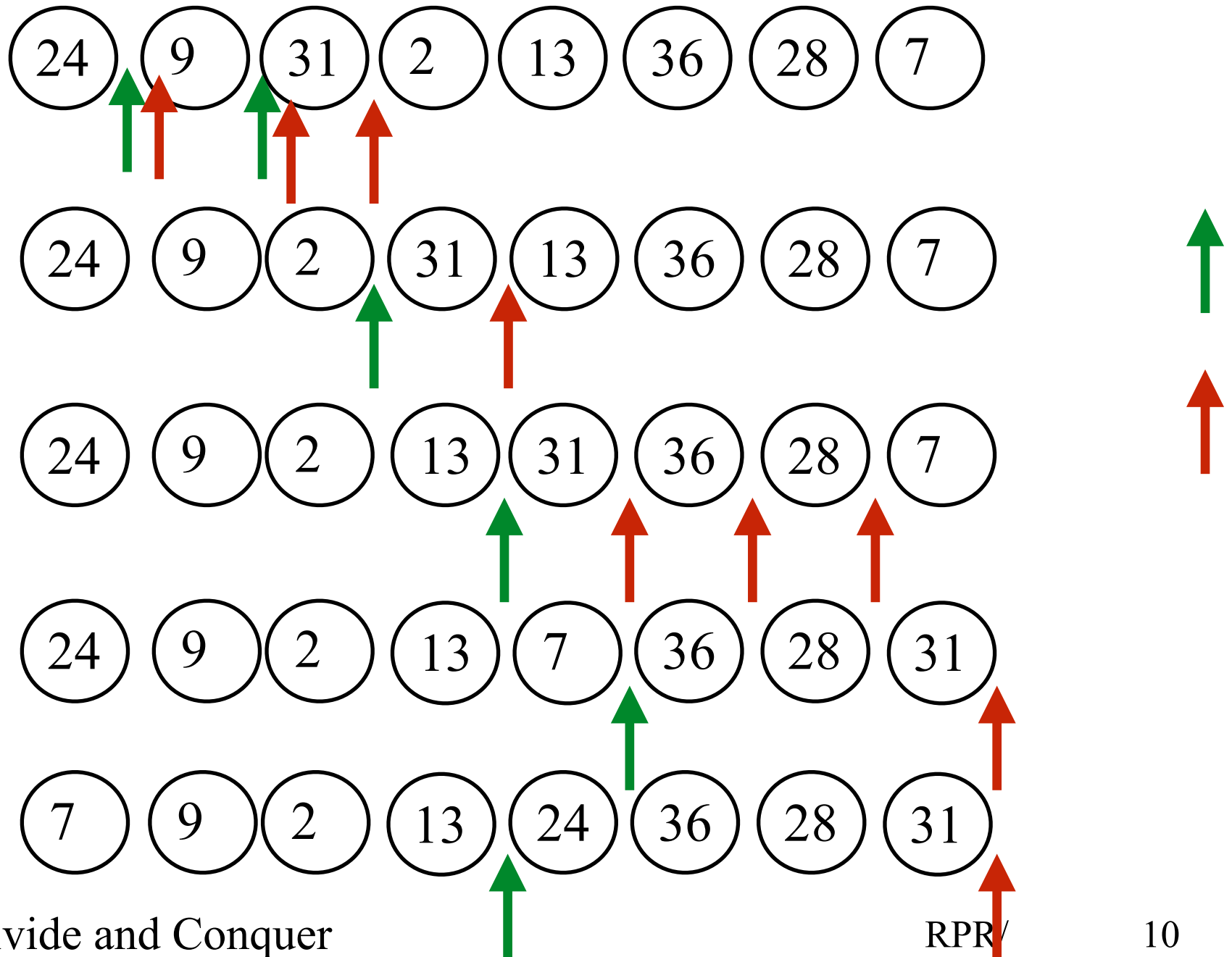
# Quicksort



- Define two pointers
  - Green: indicates end of lower partition
  - Red: end of current partitioning i.e. elements to the right are yet to be partitioned



# Quicksort



DAA/Divide and Conquer

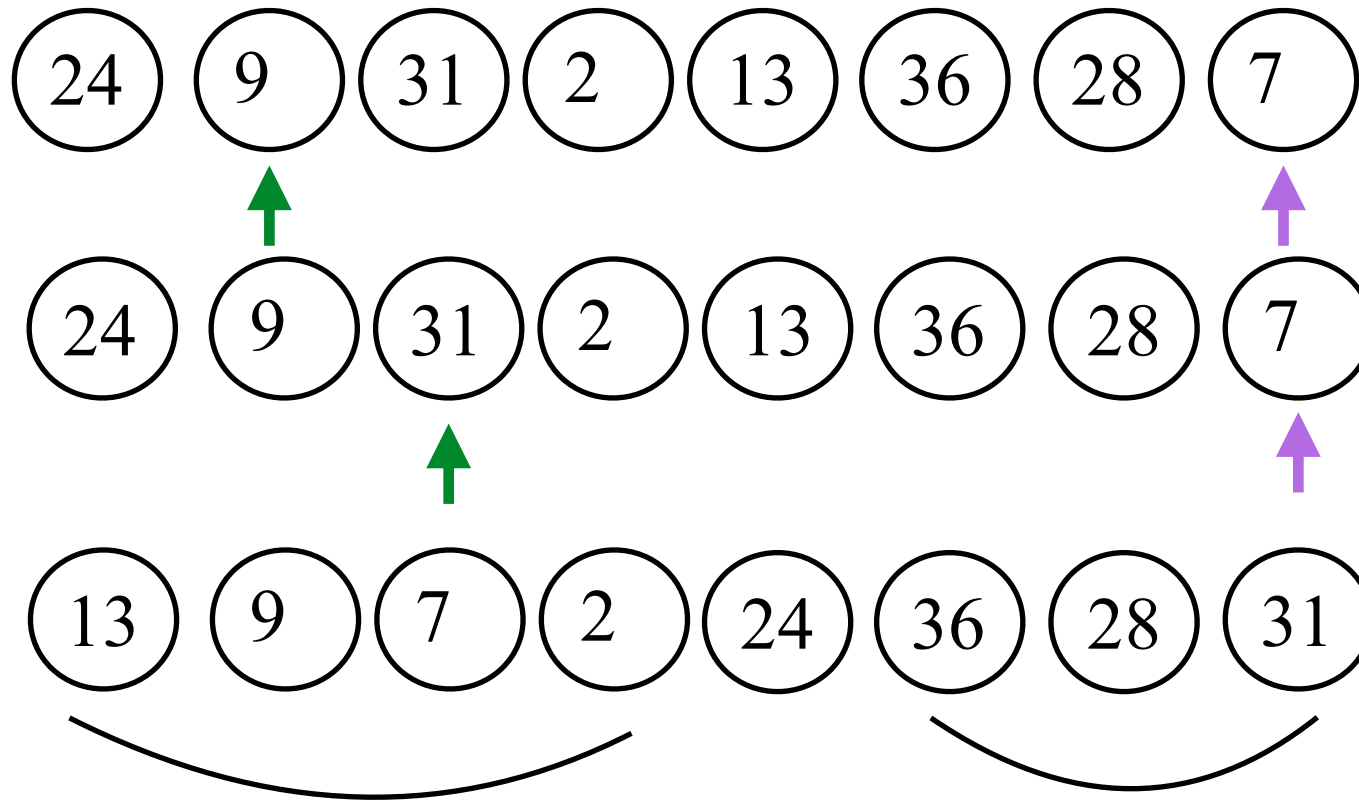
RPR/

10

# Quicksort algo

```
QSort (A, L, R) # sort A[L..R-1]
if R-L<=1 #base case
    return
#partition w.r.t. pivot
green=L+1
for red=L+1, red<R, red++)
    if A[red]<=A[L] #pivot
        swap(A[green], A[red])
        green++
swap(A[L], A[green-1]) #move pivot into place
QSort (A, L, green)
QSort (A, green+1, R)
```

# Quicksort (Another Partitioning)



# Quicksort (Book)

- **Algo** quicksort(left, right, A[])

# array index starts from 0 to n-1

#i/p: left - array index to start from

right - array index up to which to consider

array[] defined by left and right indices

#o/p: array[] sorted in ascending order

```
if left < right
```

```
    s ← partition(left, right)
```

```
    quicksort(left, s-1)
```

```
    quicksort(s+1, right)
```

```
return
```

# Quicksort

- **Algo** partition(L, R, A[])  
pivot ← A[L]; i ← L; j ← R+1  
**repeat**  
    **repeat**  
        i ← i+1  
    **until** A[i] ≥ pivot  
    **repeat**  
        j ← j-1  
    **until** A[j] ≤ pivot  
    swap(A[i], A[j])  
**until** i ≥ j  
swap(A[i], A[j]) #undo last swap when i ≥ j  
swap(A[L], A[j]) #put pivot in its place  
**return** j

# Analysis: QuickSort

- **Best case: split is approximately in the middle**

$$\begin{aligned}T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \log_2 n)\end{aligned}$$

- **Worst case: split is at the end (or beginning)**

- **e.g. sorted array**

$$\begin{aligned}T(n) &= T(n-1) + \Theta(n) \\ &= \Theta(n^2)\end{aligned}$$

- **Average case:**

$$T(n) = \Theta(n \log_2 n)$$

- **Improvements (20-25%)**

- Better pivot selection : take median
- Use insertion sort on smaller array size
- Eliminate recursion and use iteration

# Analysis: QuickSort

- Recursive calls works on two segments of the array and elements of one segment are not exchanged with elements of other segments.
- Essentially, no combination of results are required
- In practice quicksort is very fast
  - Typically, the default algorithm for in-built sort functions
    - e.g. spreadsheets
  - Programming languages use this sort for built-in sort



# Summary

- Mergesort
  - Not in place sort
- Quicksort
  - In place sort
  - Practically used on large data

# Summary

- Mergesort
  - Not in place sort
- Quicksort
  - In place sort
  - Practically used on large data