



Design and Analysis
of Algorithms I

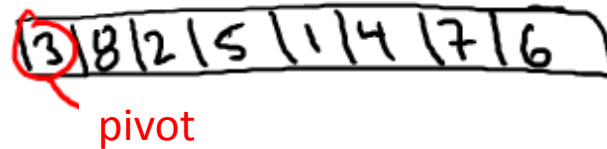
QuickSort

The Partition Subroutine

Partitioning Around a Pivot

Key Idea : partition array around a pivot element.

-Pick element of array



-Rearrange array so that

-Left of pivot => less than pivot

-Right of pivot => greater than pivot



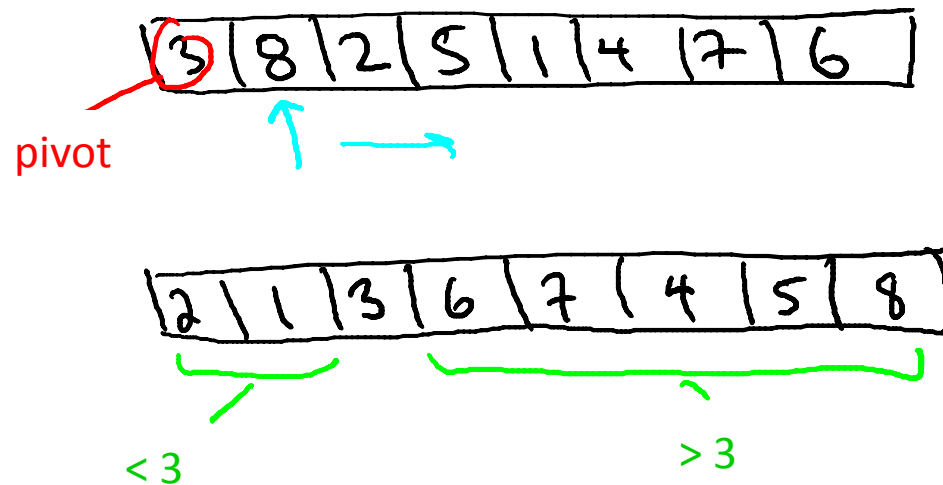
Note : puts pivot in its “rightful position”.

Two Cool Facts About Partition

1. Linear $O(n)$ time, no extra memory
[see next video]
2. Reduces problem size

The Easy Way Out

Note : Using $O(n)$ extra memory, easy to partition around pivot in $O(n)$ time.

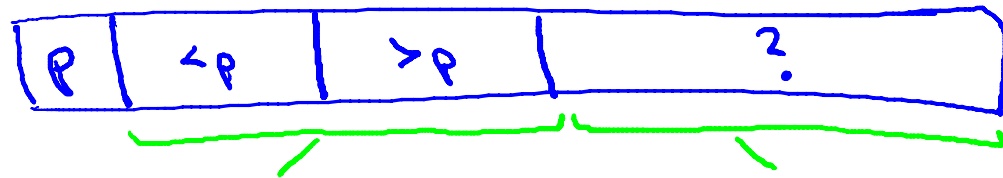


In-Place Implementation

Assume : pivot = 1st element of array

[if not, swap pivot \leftrightarrow 1st element as preprocessing step]

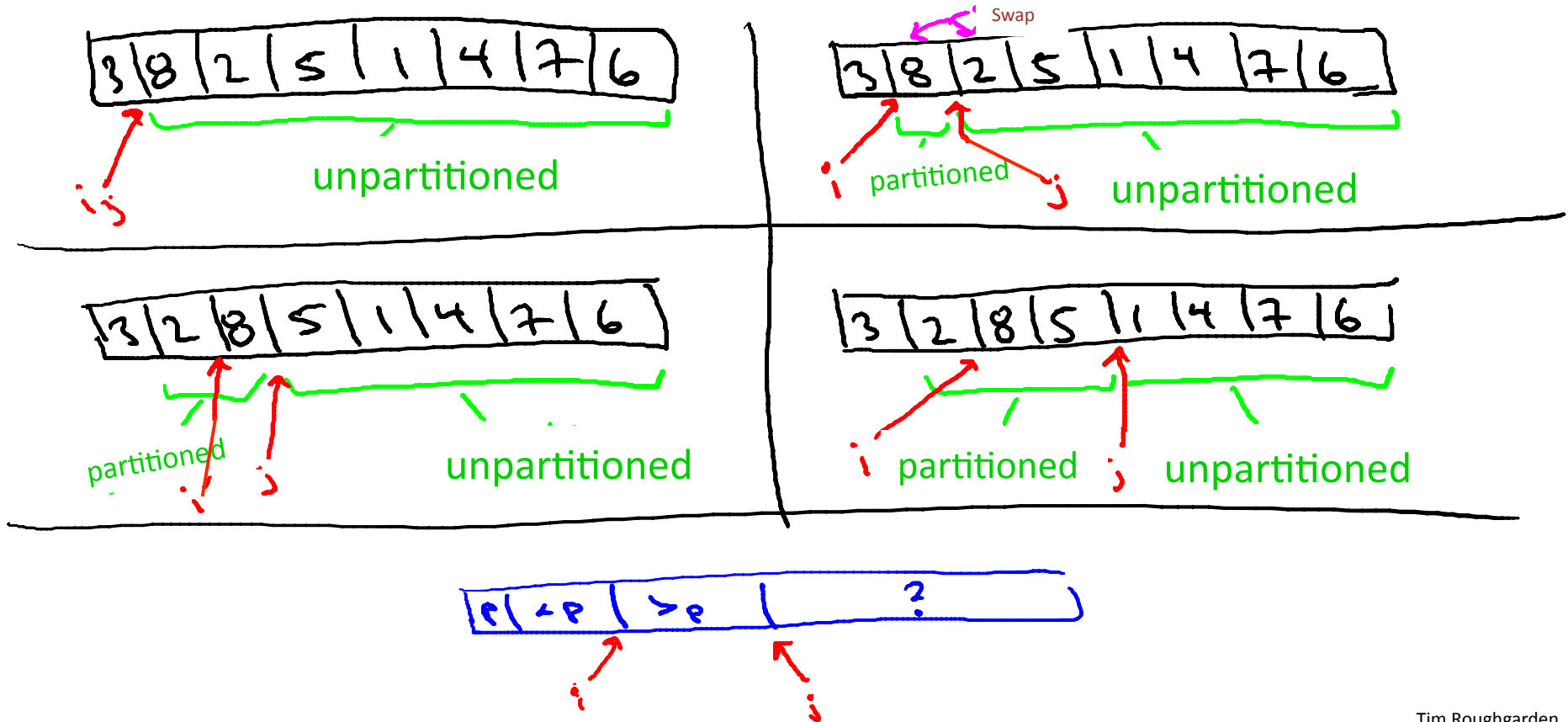
High – Level Idea :



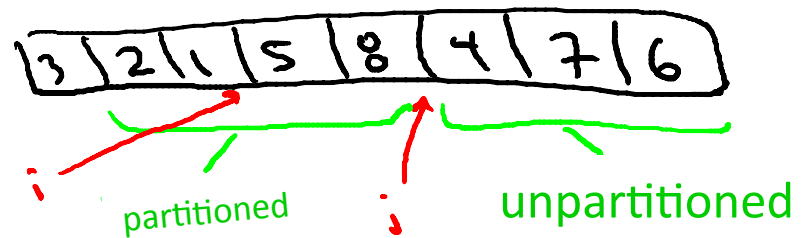
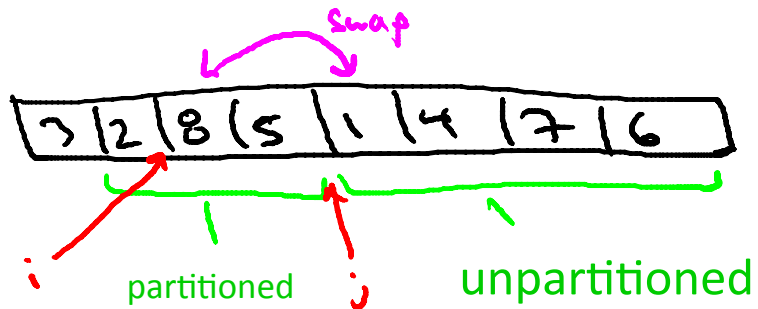
Already partitioned unpartitioned

- Single scan through array
- invariant : everything looked at so far is partitioned

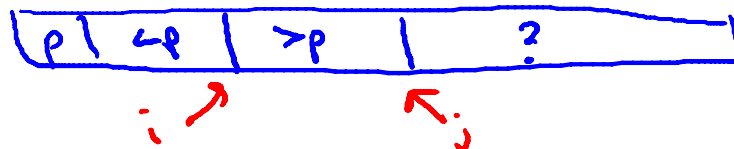
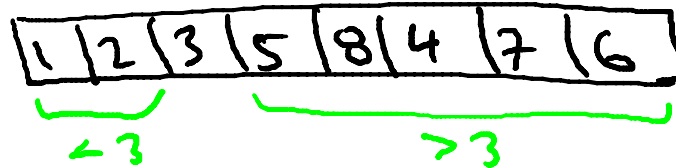
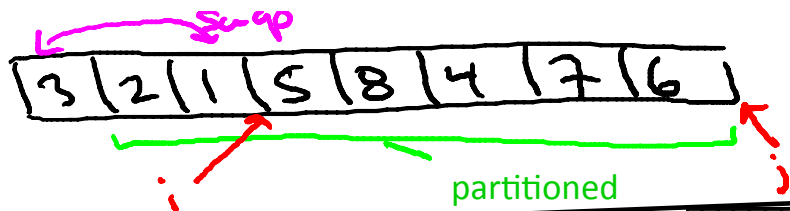
Partition Example



Partition Example (con'd)



Fast forwarding



Pseudocode for Partition

Partition (A,l,r)

[input corresponds to A[l...r]]

- $p := A[l]$

- $i := l+1$

- for $j=l+1$ to r

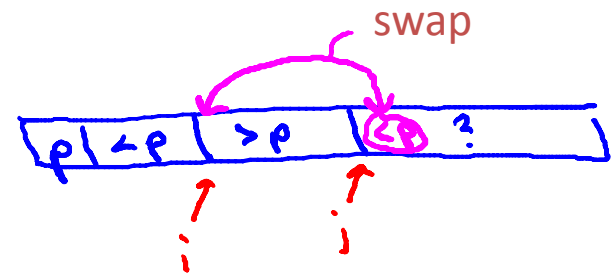
- if $A[j] < p$

[if $A[j] > p$, do nothing]

- swap $A[j]$ and $A[i]$

- $i := i+1$

- swap $A[l]$ and $A[i-1]$



Running Time

Running time = $O(n)$, where $n = r - l + 1$ is the length of the input (sub) array.

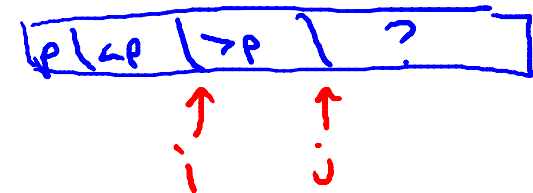
Reason : $O(1)$ work per array entry.

Also : clearly works in place (repeated swaps)

Correctness

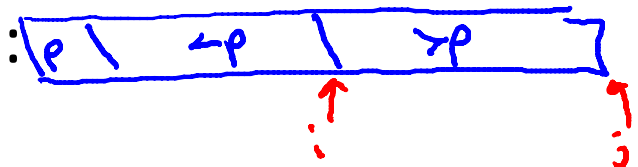
Claim : the for loop maintains the invariants :

1. $A[l+1], \dots, A[i-1]$ are all less than the pivot
2. $A[i], \dots, A[j-1]$ are all greater than pivot.



[Exercise : check this, by induction.]

Consequence : at end of for loop, have:



=> after final swap, array partitioned around pivot.

Q.E.D