# Quick Sort

- To sort a subarray A[p..r]
  - Choose an element from the subarray as a *pivot*
- Divide:
  - Partition the array A[p..r] into two (possibly empty) subarrays A[p.. q-1] and A[q+1..r] such that each element of A[p..q-1] <= A[q] (pivot) and each element of A[q+1..r] >= A[q] (pivot)
- Conquer
  - Recursively sort the two subarrays A[p..q-1], A[q+1..r]
- Combine: no work needs to be done.

# The algorithm

QUICKSORT($A, p, r$)

**if** $p < r$

  **then** $q \leftarrow$ PARTITION($A, p, r$)

    QUICKSORT($A, p, q - 1$)

    QUICKSORT($A, q + 1, r$)

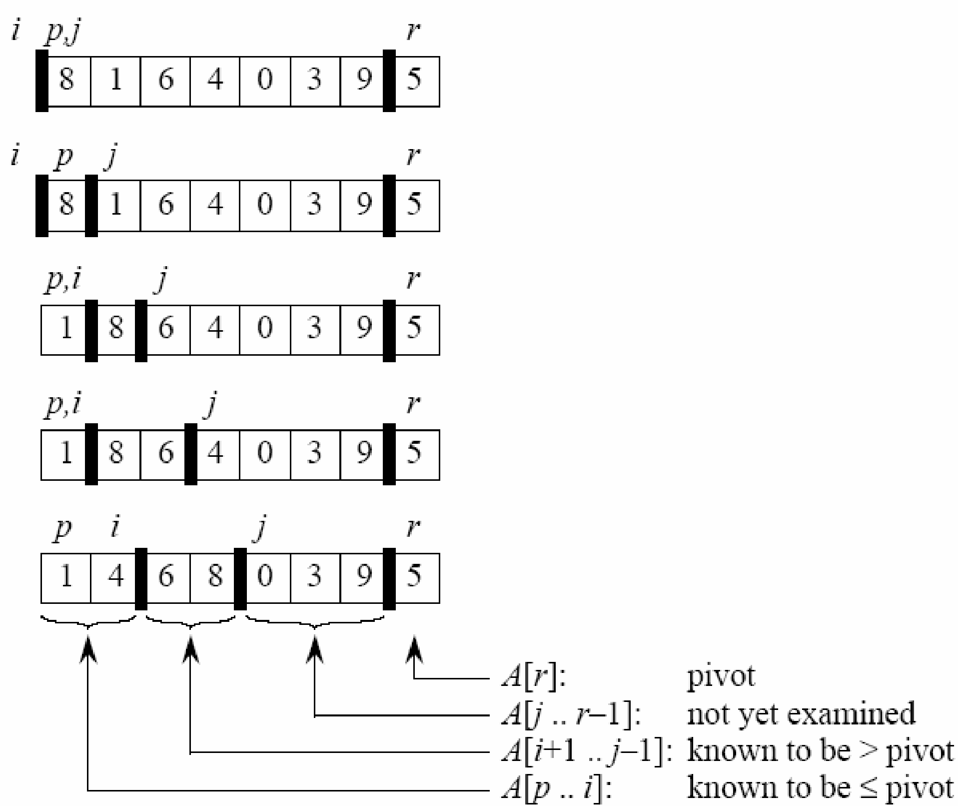
Initial call is QUICKSORT($A, 1, n$).

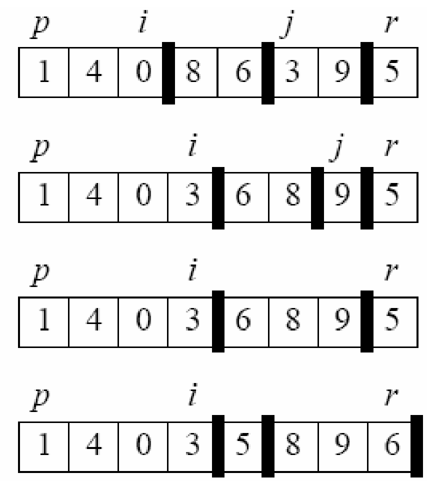
# Partition

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
     do if A[j] ≤ x
          then i ← i + 1
               exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

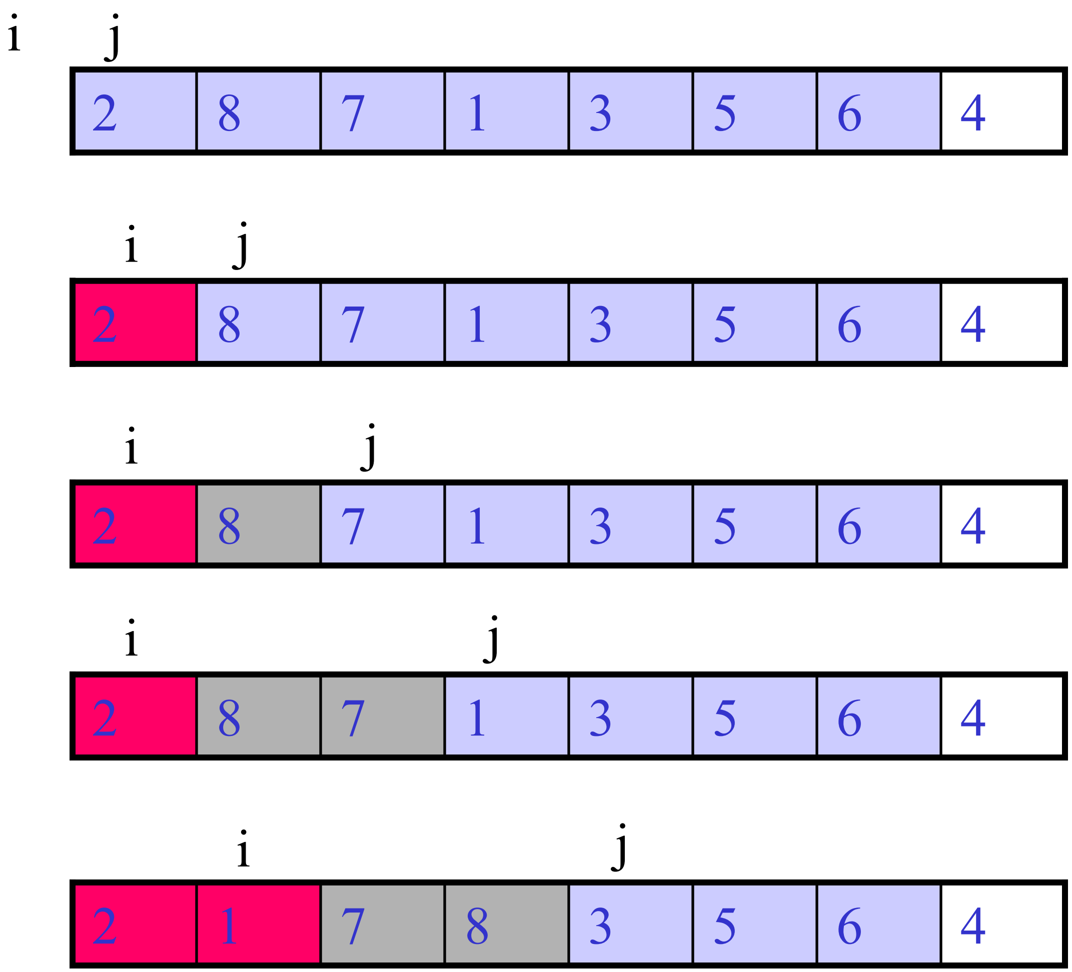
- always selects the last element A[r] as pivot – the element around which to partition
- as procedure executes, array is partitioned into four regions

# Example for Partition

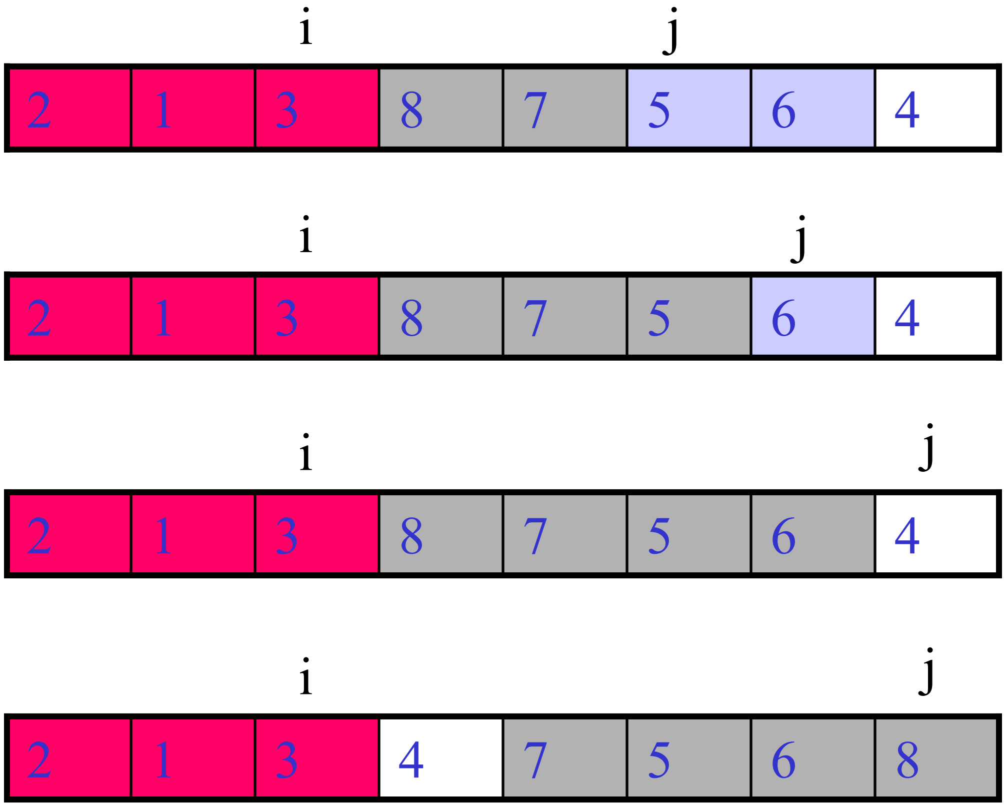


# Example for Partition



# Example for Partition




State when loop finishes

Final swap after loop

Time for partitioning: $\Theta(n)$

## Loop invariant

- All entries in A[p..i] are <= pivot

- All entries in A[i+1..j-1] > pivot

- A[r]=pivot

# Analysis

Worst case:

- the array is sorted
- 0 elements in one subarray and n-1 elements in other
- $T(n)=T(n-1)+T(0)+\Theta(n)$

  $= T(n-1)+\Theta(n)$

  $= \Theta(n^2)$
- Same running time as insertion sort

# Analysis

- Best case
  - Each subarray has <= n/2 elements
  - $T(n) = 2T(n/2) + \Theta(n)$

    $= \Theta(n \log n)$

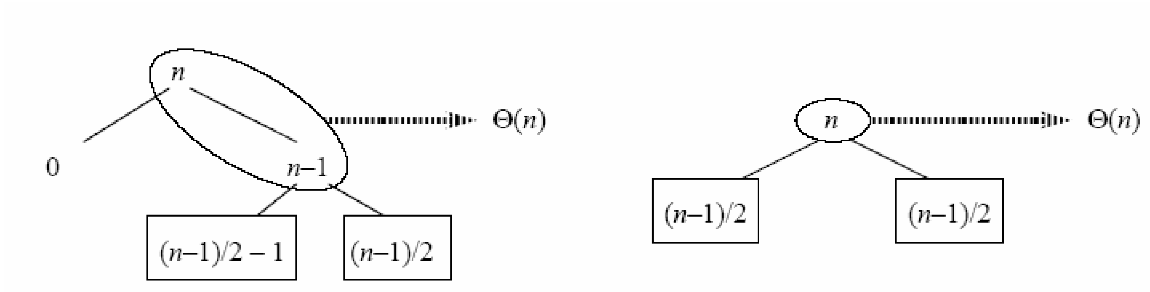
# Analysis

- Balanced partitioning
  - Quicksort's average running time is much closer to best case than to worst case
  - Imagine that Partition always produces a 9-to-1 split
  - $T(n) \le T(9n/10) + T(n/10) + \Theta(n)$

    $= O(n \log n)$

# Analysis

- Intuition for average case
  - Splits in recursion tree will not always be constant
  - There wil lbe a mix of good and bad splits
  - This doesn't affect the asyptotic running tine of quicksort




  - Extra level in left-hand-side only adds to constant in Θ
  - Still the same number of subarrays to sort, only twice as much work to get there
  - Both figures results in O(n log n) time, different constant

# Randomized Quicksort

- it is not always true that all input permutation are equally likely
- add randomization to quicksort
- could randomly permute the input array
- instead, use random sampling: pick one element at random
- don't always use A[r] as pivot, randomly pick one
- on average, this cause the split of the input to be reasonably well balanced

```
RANDOMIZED-PARTITION(A, p, r)
i ← RANDOM(p, r)
exchange A[r] ↔ A[i]
return PARTITION(A, p, r)
```

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