Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 8

Divide-and-Conquer

- **Divide** the problem into a number of subproblems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough $\Rightarrow$ solve the problems in straightforward manner
- **Combine** the solutions to the sub-problems
  - Obtain the solution for the original problem

Analyzing Divide and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size n
  - **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
  - **Conquer** (solve) the subproblems: takes $aT(n/b)$
  - **Combine** the solutions: takes $C(n)$

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

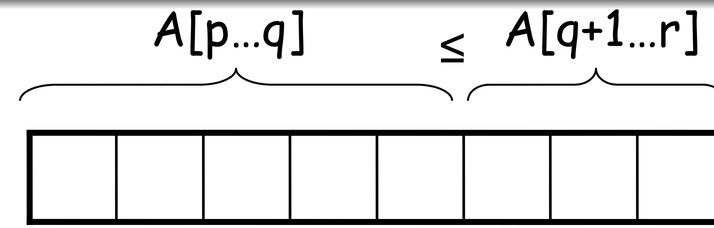
Merge Sort Approach

- To sort an array $A[p\,..\,r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Merge Sort - Discussion

- Running time insensitive of the input
- Advantages:
  - Guaranteed to run in $\Theta(n\lg n)$
- Disadvantage
  - Requires extra space ≈N
- Applications
  - Maintain a large ordered data file
  - How would you use Merge sort to do this?

Quicksort



- Sort an array *A[p…r]*
- **Divide**
  - Partition the array *A* into 2 subarrays *A[p..q]* and *A[q+1..r]*, such that each element of *A[p..q]* is smaller than or equal to each element in *A[q+1..r]*
  - The index (pivot) *q* is computed
- **Conquer**
  - Recursively sort *A[p..q]* and *A[q+1..r]* using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place ⇒ no work needed to combine them: the entire array is now sorted

QUICKSORT

Alg.: QUICKSORT(A, p, r)

if p < r

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

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

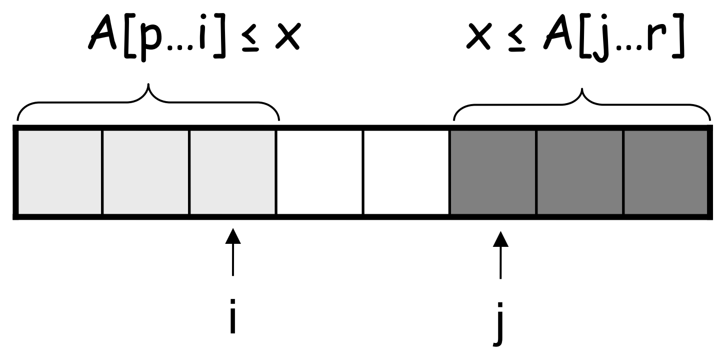
Partitioning the Array

- Idea
  - Select a pivot element **x** around which to partition
  - Grows two regions

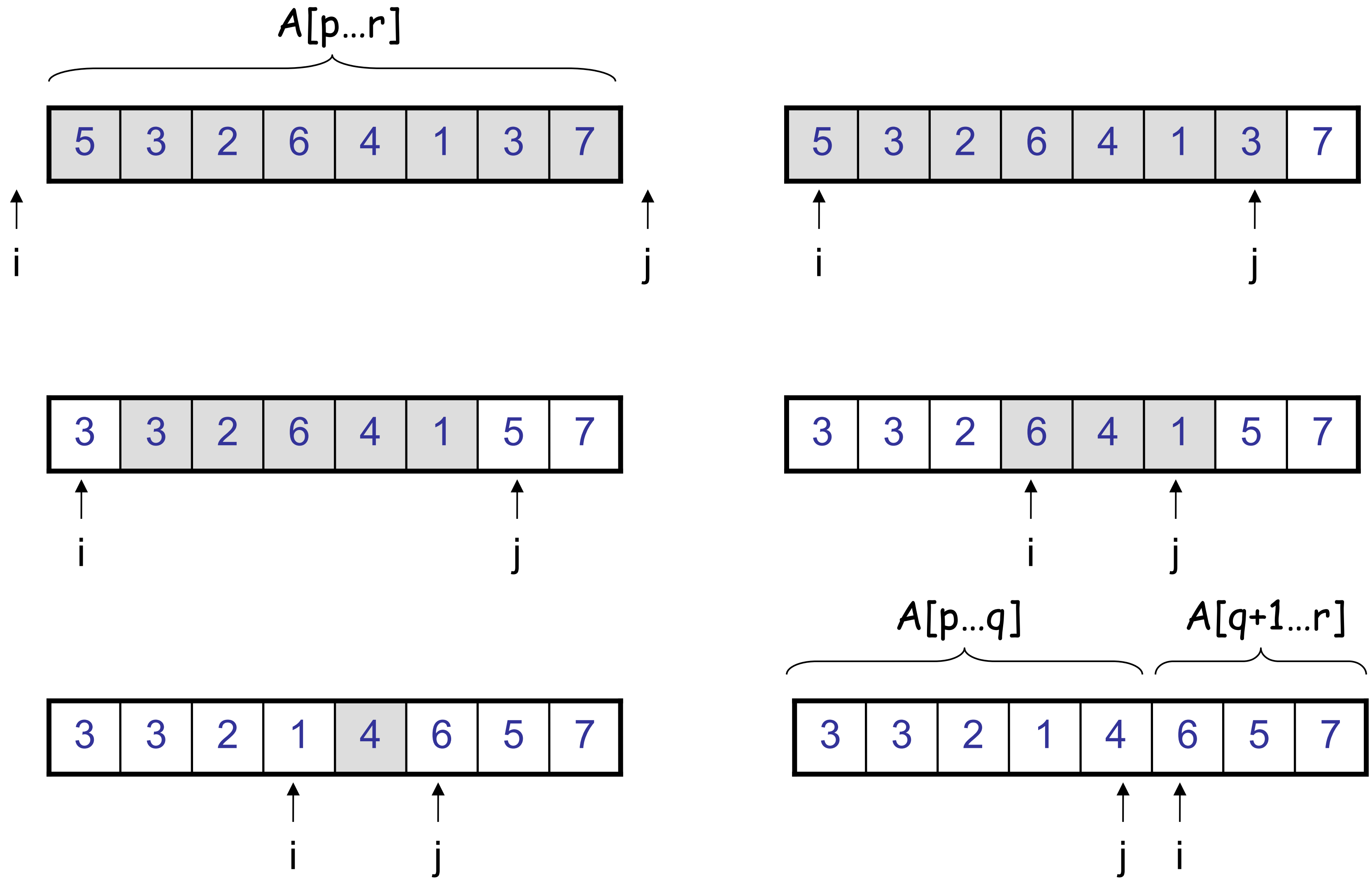
    $A[p\ldots i] \le x$

    $x \le A[j\ldots r]$

  - For now, choose the value of the first element as the pivot **x**



Example

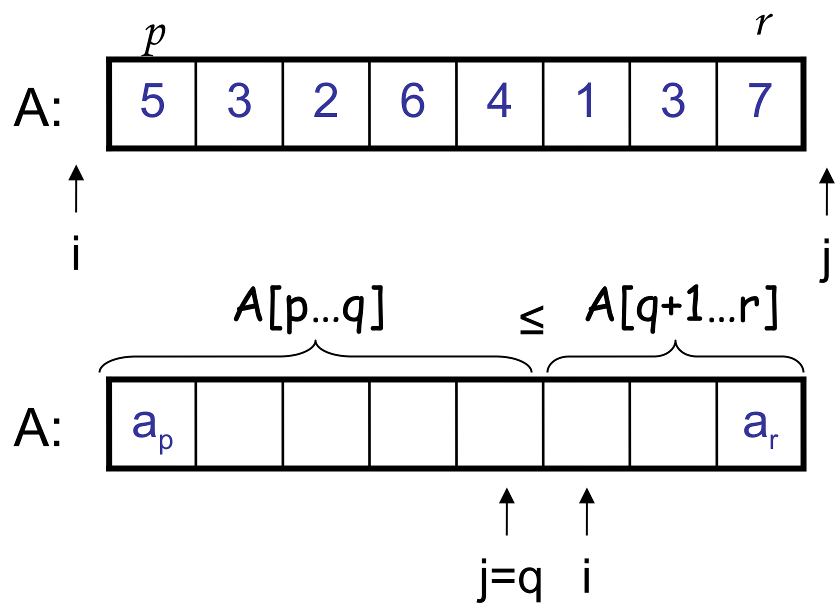


A[p...r]
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 1 4 6 5 7
i
j
A[p...q]
A[q+1...r]
3 3 2 1 4 6 5 7
j
i

Partitioning the Array

Alg. PARTITION (A, p, r)

```
x ← A[p]
i ← p - 1
j ← r + 1
while TRUE
        do repeat j ← j - 1
              until A[j] ≤ x
           repeat i ← i + 1
              until A[i] ≥ x
           if i < j
              then exchange A[i] ⟺ A[j]
                else return j
```



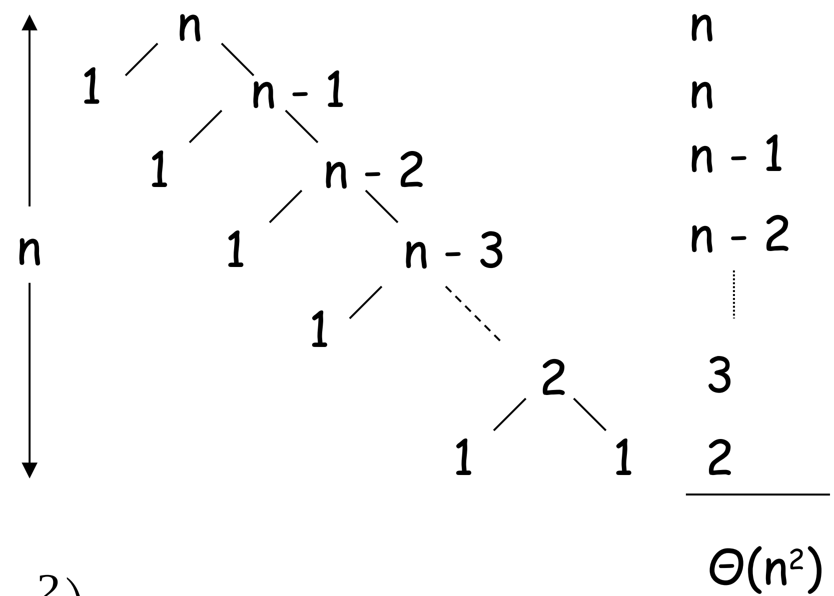
Running time: $\Theta(n)$

$n = r - p + 1$

Performance of Quicksort

- Worst-case partitioning
  - One region has 1 element and one has n – 1 elements
  - Maximally unbalanced
- Recurrence

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



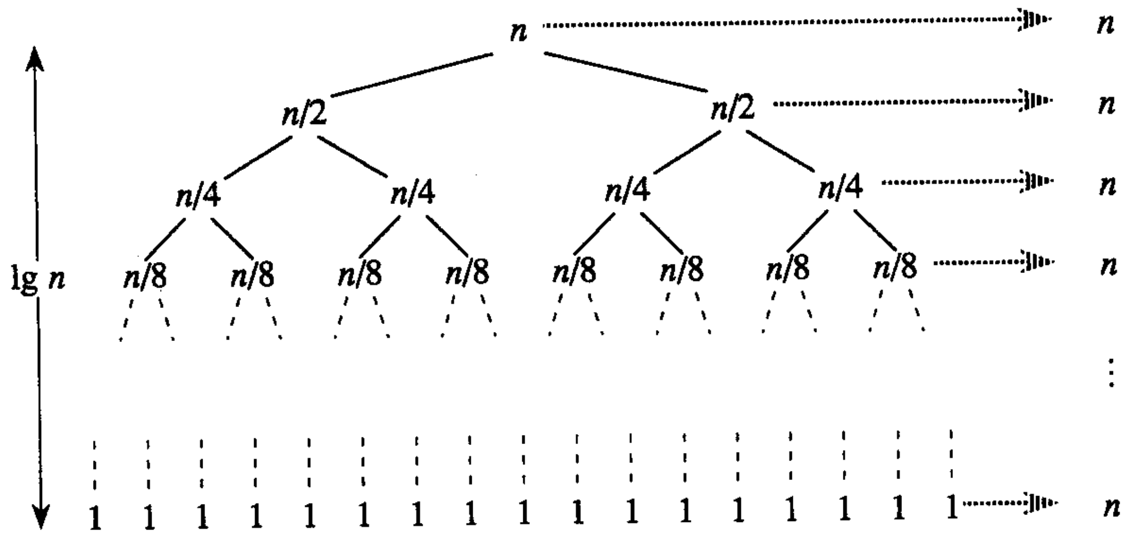
$$= n + \left(\sum_{k=1}^{n} k \right) - 1 = \theta(n^2)$$

Performance of Quicksort

- Best-case partitioning
  - Partitioning produces two regions of size $n/2$
- Recurrence

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

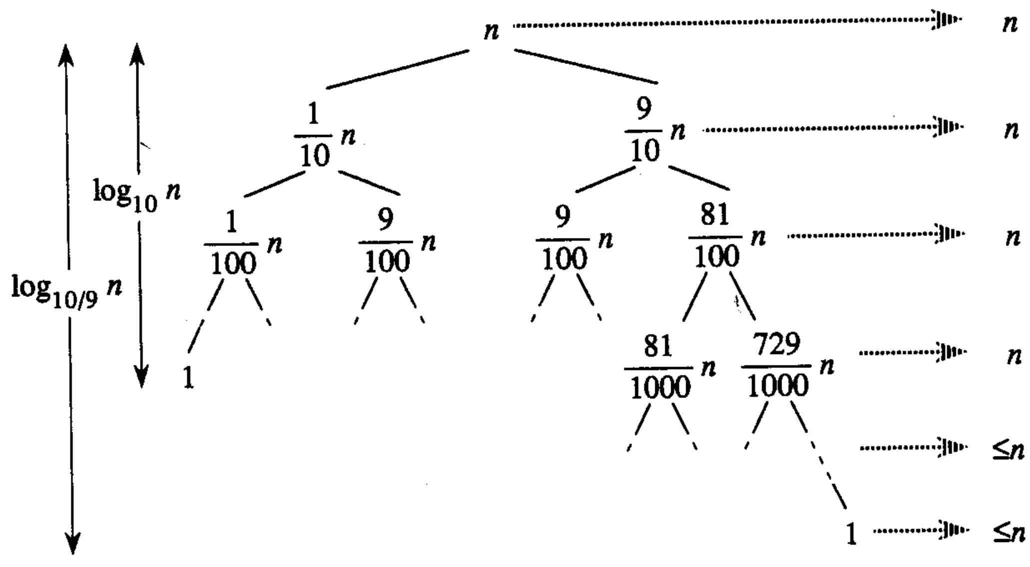
  $T(n) = \Theta(n\lg n)$ (Master theorem)



Performance of Quicksort

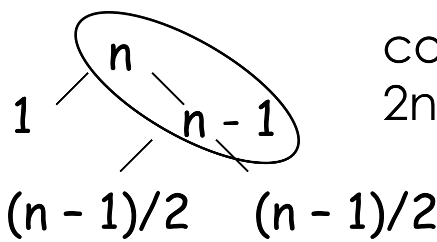
- Balanced partitioning
  - Average case is closer to best case than to worst case
  - (if partitioning always produces a **constant** split)
- E.g.: 9-to-1 proportional split

$$T(n) = T(9n/10) + T(n/10) + n$$



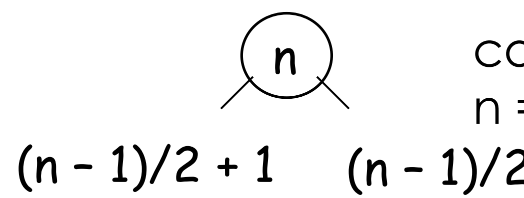
Performance of Quicksort

- Average case
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a mix of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed throughout the tree



combined cost:
$2n-1 = \Theta(n)$

Alternation of a bad and a good split



combined cost:
$n = \Theta(n)$

Nearly well balanced split

- Running time of Quicksort when levels alternate between good and bad splits is O(nlgn)

Worst-Case Analysis of Quicksort

- $T(n)$ = worst-case running time
- $T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n)$
- Use substitution method to show that the running time of Quicksort is $O(n^2)$
- Guess $T(n) = O(n^2)$
  - Induction goal: $T(n) \le cn^2$
  - Induction hypothesis: $T(k) \le ck^2$ for any $k \le n$

Worst-Case Analysis of Quicksort

- Proof of induction goal:

$$T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) =$$

$$= c \times \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n)$$

- The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at the endpoints of this interval

$$\max_{1 \le q \le n-1} (q^2 + (n - q)^2) = 1^2 + (n - 1)^2 = n^2 - 2(n - 1)$$

$$T(n) \le cn^2 - 2c(n - 1) + \Theta(n)$$

$$\le cn^2$$

All material up to this point will be included for midterm 1
February 27, during class

BREAKPOINT FOR MIDTERM 1

Randomizing Quicksort

- Randomly permute the elements of the input array before sorting
- Modify the PARTITION procedure
  - First we exchange element $A[p]$ with an element chosen at random from $A[p \ldots r]$
  - Now the pivot element $x = A[p]$ is equally likely to be any one of the original $r - p + 1$ elements of the subarray

Randomized Algorithms

- The behavior is determined in part by values produced by a random-number generator
  - RANDOM(**a, b**) returns an integer **r**, where $a \le r \le b$ and each of the **b-a+1** possible values of **r** is equally likely
- Algorithm generates randomness in input
- No input can consistently elicit worst case behavior
  - Worst case occurs only if we get "unlucky" numbers from the random number generator

Randomized PARTITION

Alg.: RANDOMIZED-PARTITION(A, p, r)

i ← RANDOM(p, r)

exchange A[p] ⟷ A[i]

return PARTITION(A, p, r)

Randomized Quicksort

Alg. : RANDOMIZED-QUICKSORT(A, p, r)

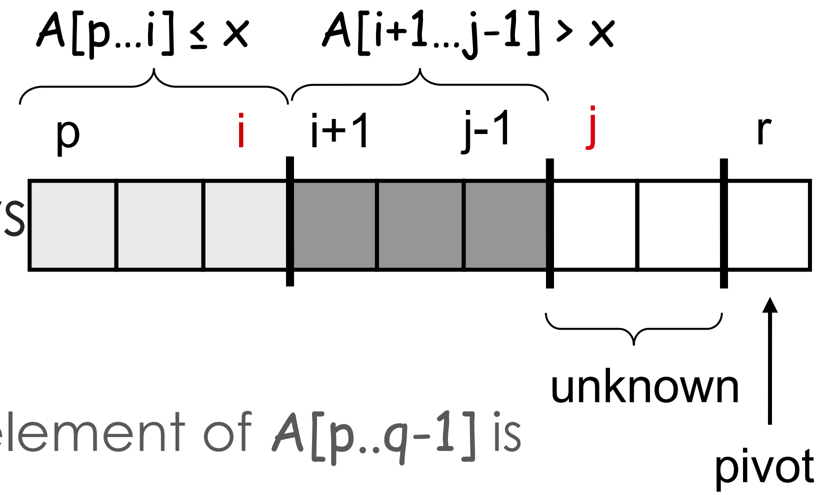
if p < r

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

RANDOMIZED-QUICKSORT(A, p, q)

RANDOMIZED-QUICKSORT(A, q + 1, r)

Another Way to PARTITION



- Given an array A, partition the array into the following subarrays
  - A pivot element $x = A[q]$
  - Subarray $A[p..q-1]$ such that each element of $A[p..q-1]$ is smaller than or equal to x (the pivot)
  - Subarray $A[q+1..r]$, such that each element of $A[p..q+1]$ is strictly greater than x (the pivot)
- Note: the pivot element is not included in any of the two subarrays

Another Way to PARTITION

Alg.: PARTITION2(*A, p, r)*

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

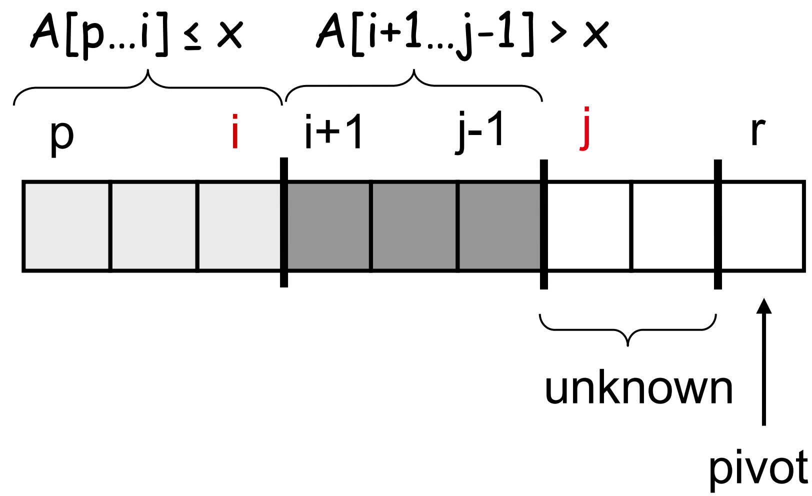
do if $A[j] \leq x$

then $i \leftarrow i + 1$

exchange $A[i] \leftrightarrow A[j]$

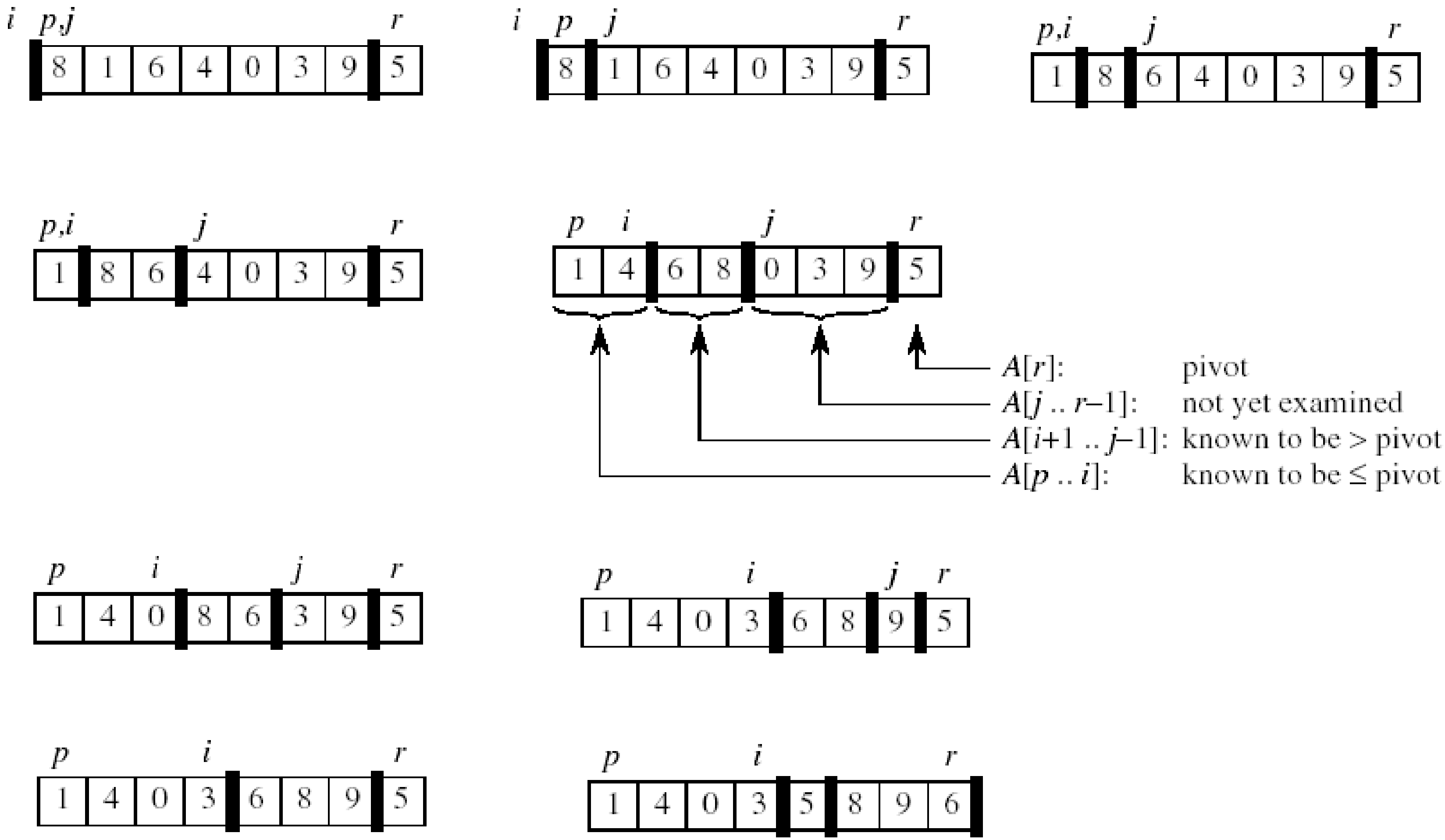
exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$



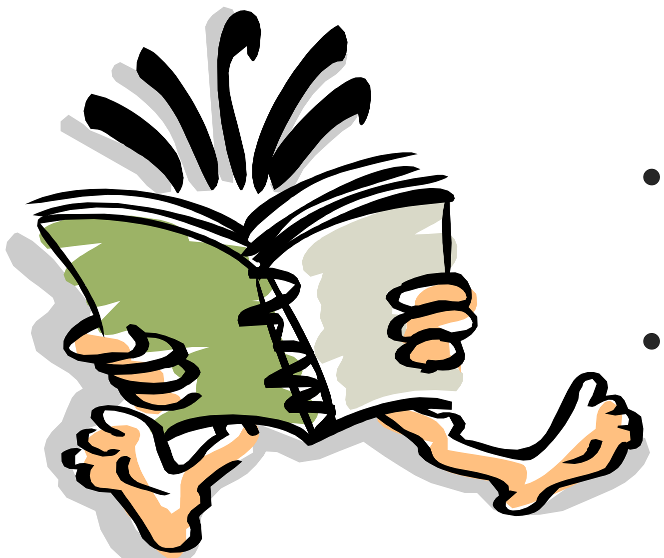
- Chooses the last element of the array as a pivot
- Grows a subarray [p..i] of elements ≤ x
- Grows a subarray [i+1..j-1] of elements >x
- Running Time: $\Theta(n)$, where $n=r-p+1$

Example



i p,j r
8 1 6 4 0 3 9 5
i p j r
8 1 6 4 0 3 9 5
p,i j r
1 8 6 4 0 3 9 5
p,i j r
1 8 6 4 0 3 9 5
p i j r
1 4 6 8 0 3 9 5
A[r]: pivot
A[j .. r–1]: not yet examined
A[i+1 .. j–1]: known to be > pivot
A[p .. i]: known to be ≤ pivot
p i j r
1 4 0 8 6 3 9 5
p i j r
1 4 0 3 6 8 9 5
p i r
1 4 0 3 6 8 9 5
p i r
1 4 0 3 5 8 9 6

Readings



- For this lecture
  - Section 7.2-7.4
- Coming next
  - Chapter 9