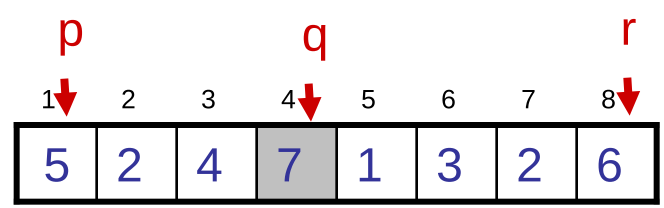
Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  – 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 of the sub-problems
  – Obtain the solution for the original problem

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



Alg.: MERGE-SORT(A, p, r)

if p < r ▷ Check for base case

then $q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

MERGE-SORT(A, p, q) ▷ Conquer

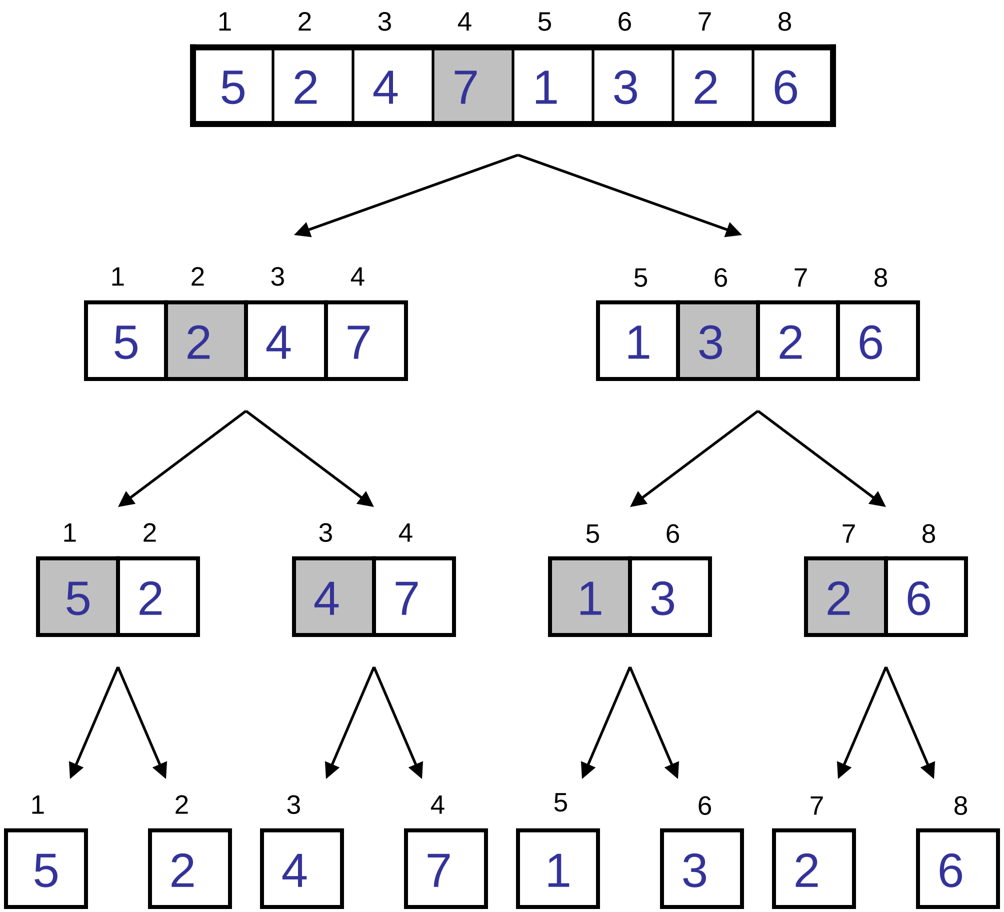
MERGE-SORT(A, q + 1, r) ▷ Conquer

MERGE(A, p, q, r) ▷ Combine

- Initial call: MERGE-SORT(A, 1, n)

Example – *n* Power of 2

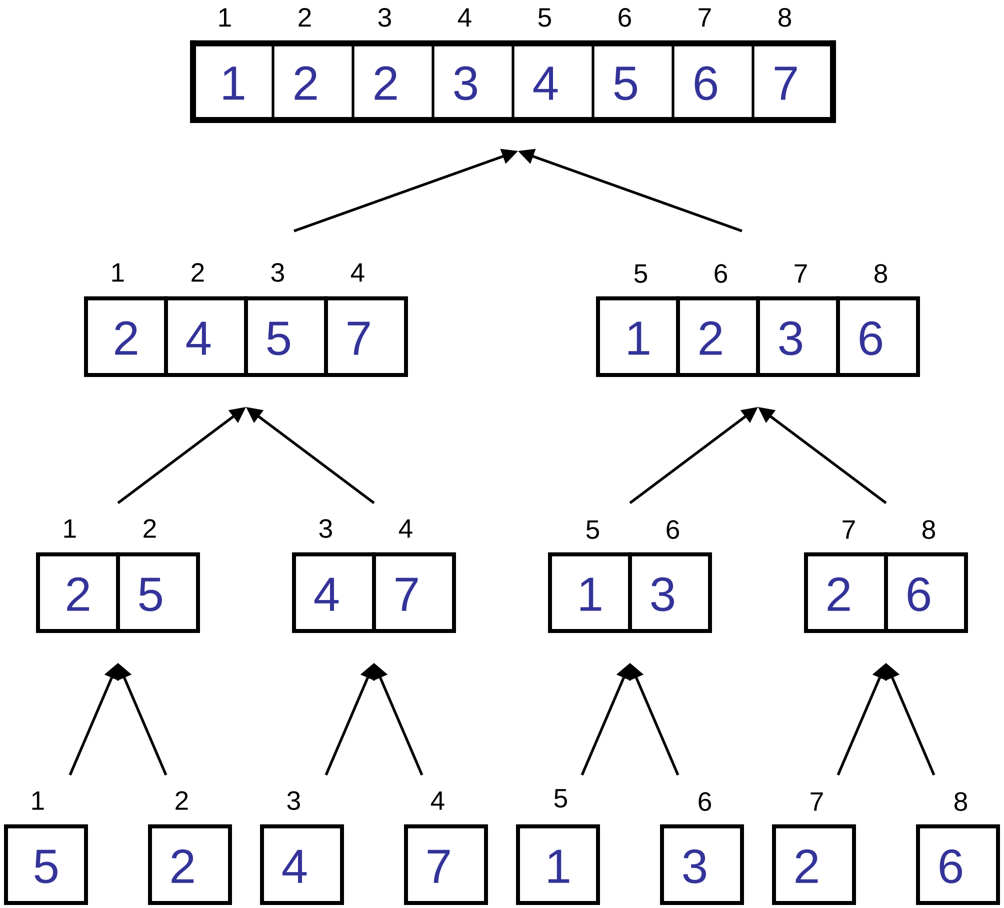
Divide



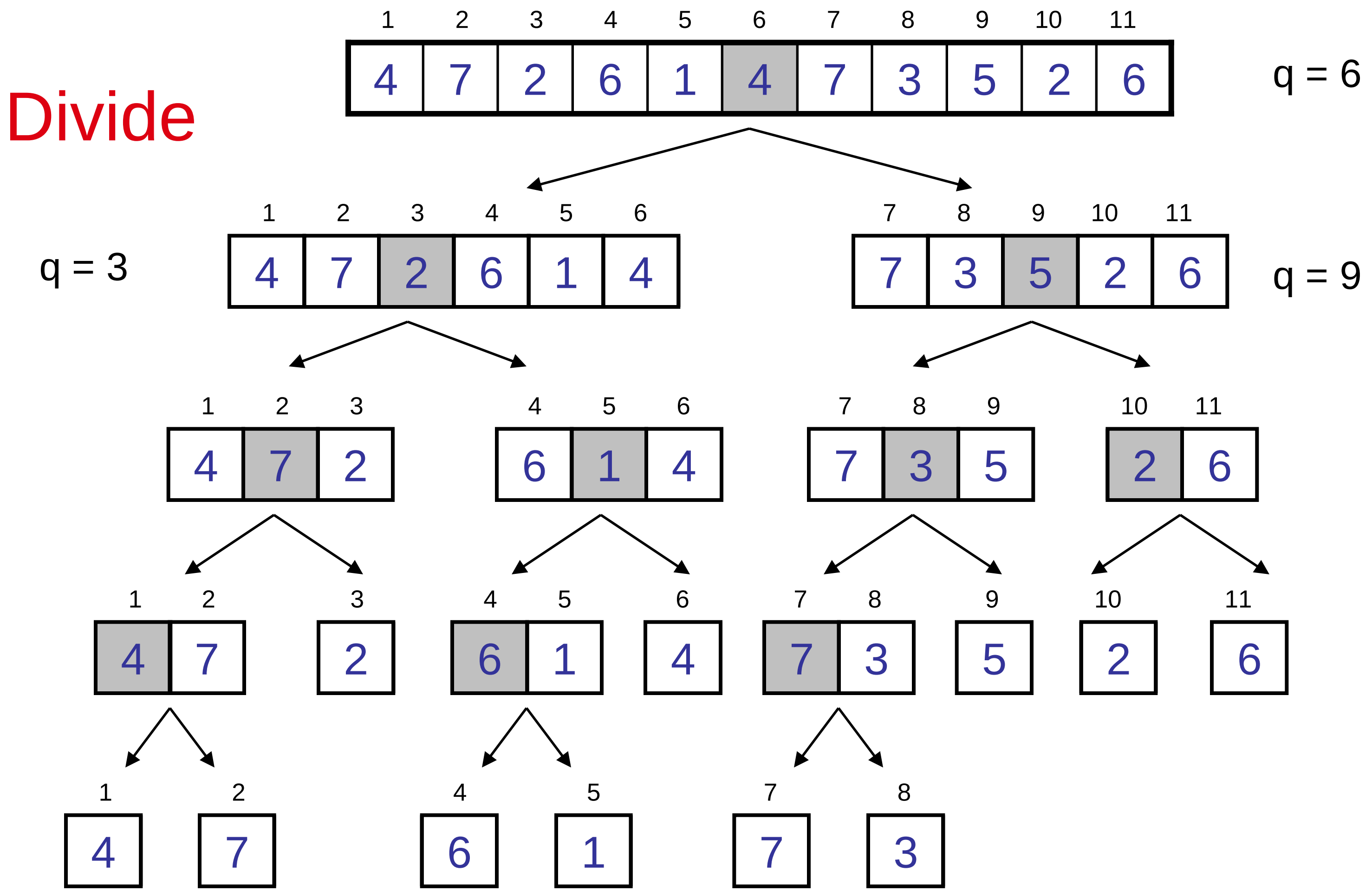
q = 4

Example – n Power of 2

Conquer and Merge

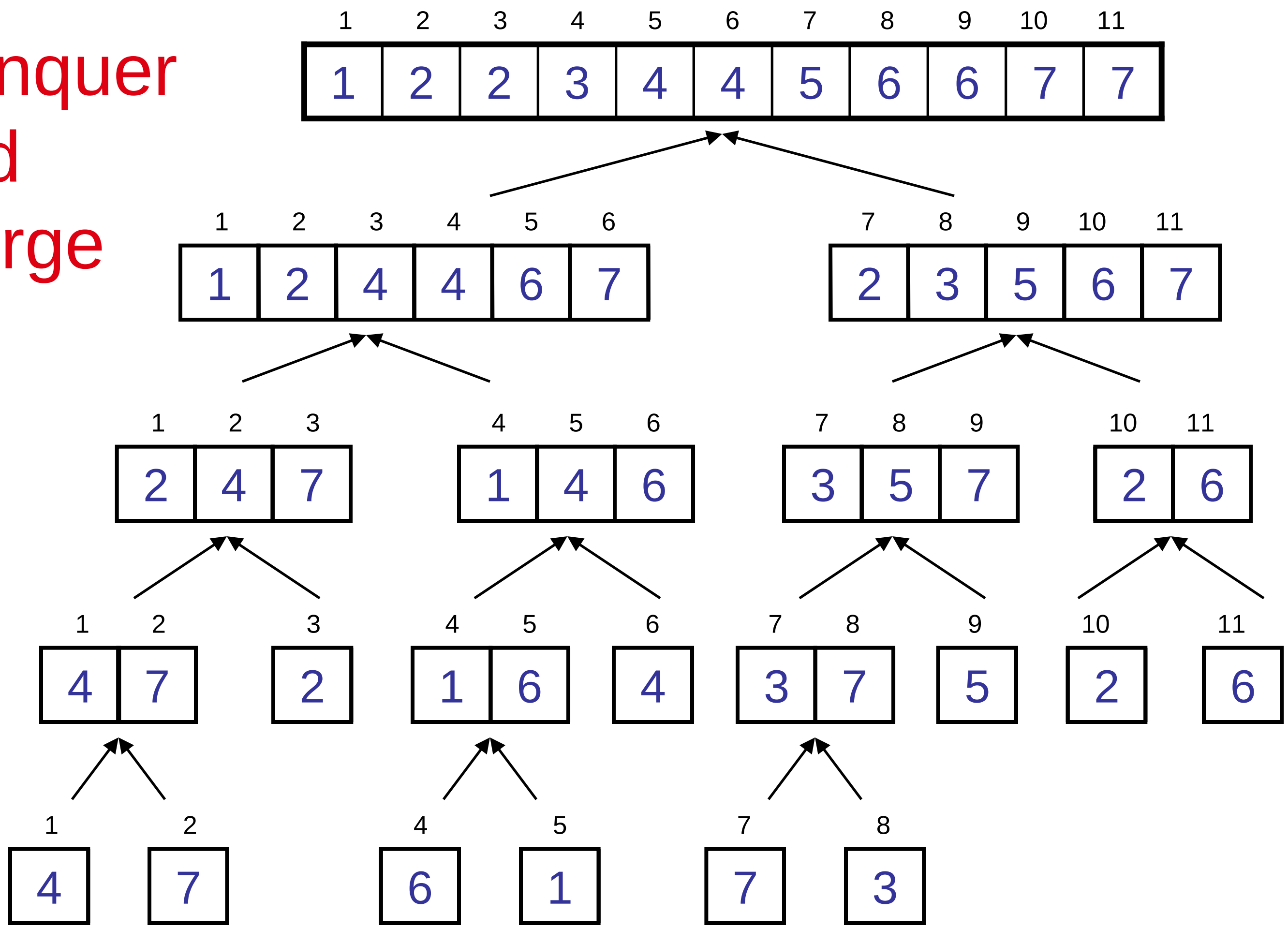


Example – *n* Not a Power of 2

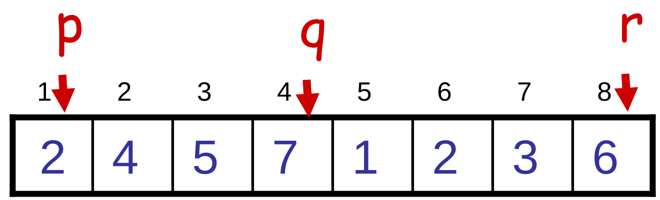


Example – n Not a Power of 2

Conquer and Merge



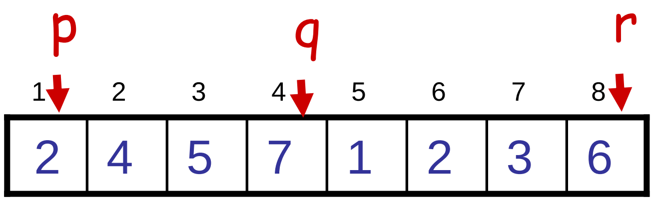
Merging



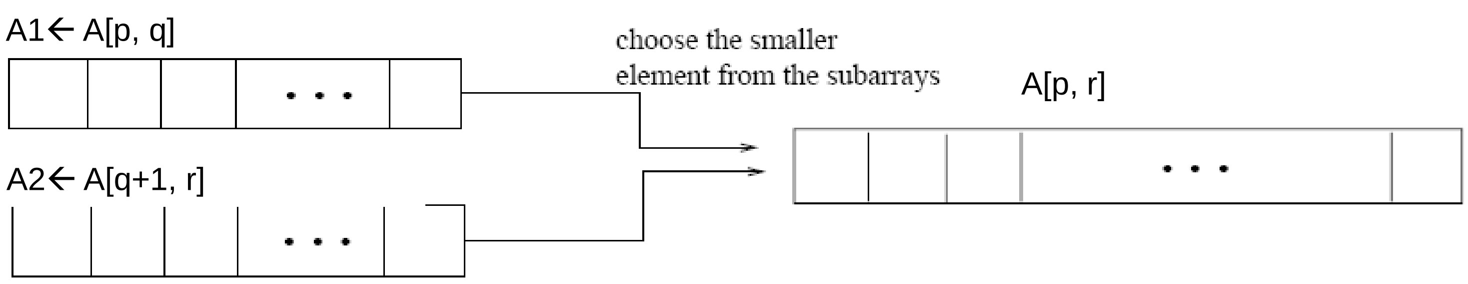
- **Input:** Array A and indices p, q, r such that $p \le q < r$
  - Subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ are sorted
- **Output:** One single sorted subarray $A[p \,..\, r]$

Merging

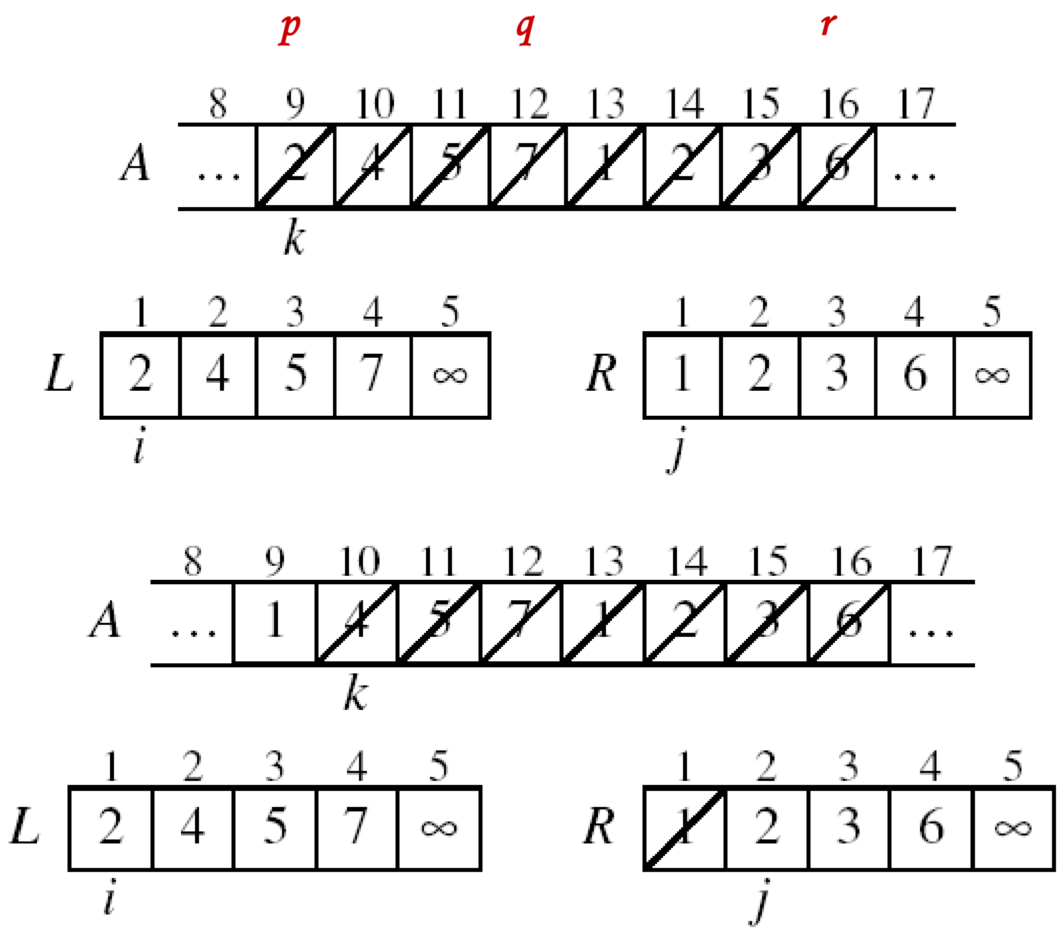
- Idea for merging:



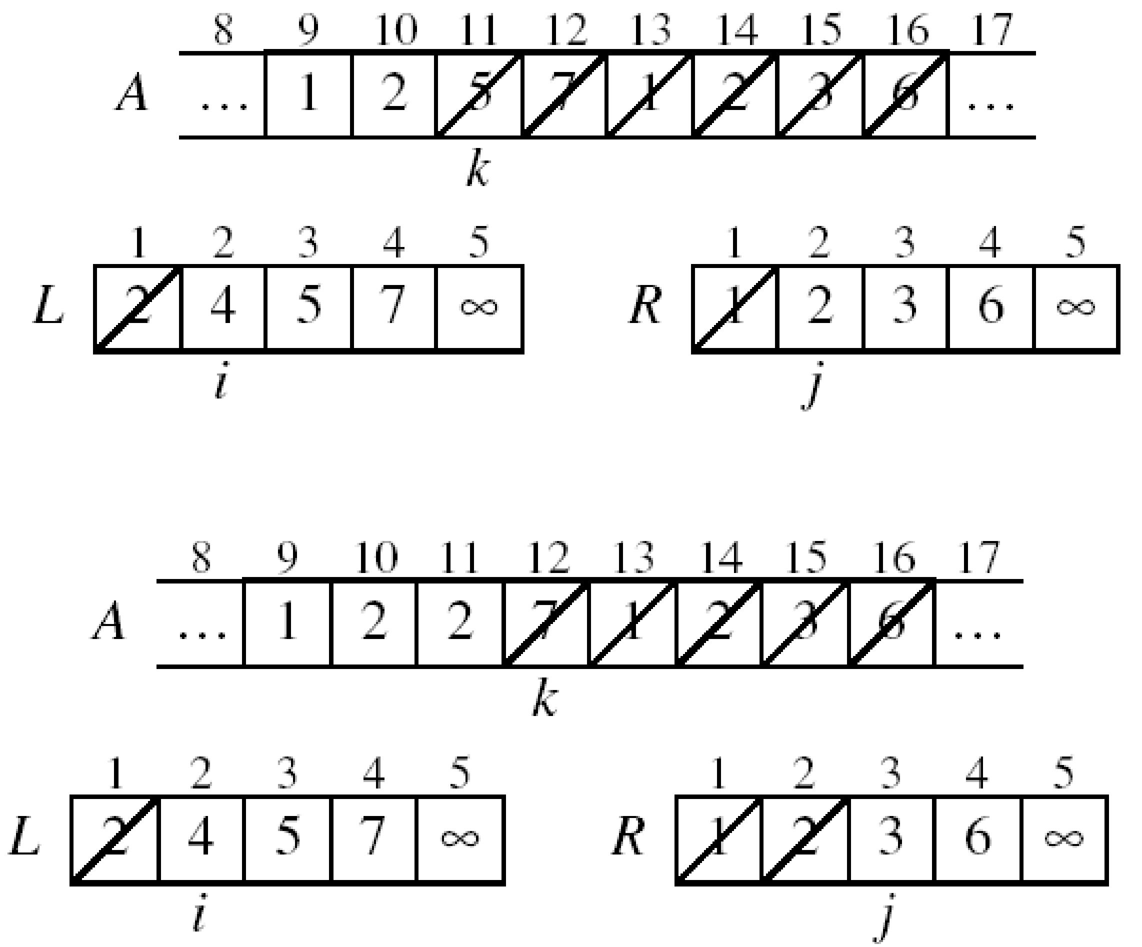
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile



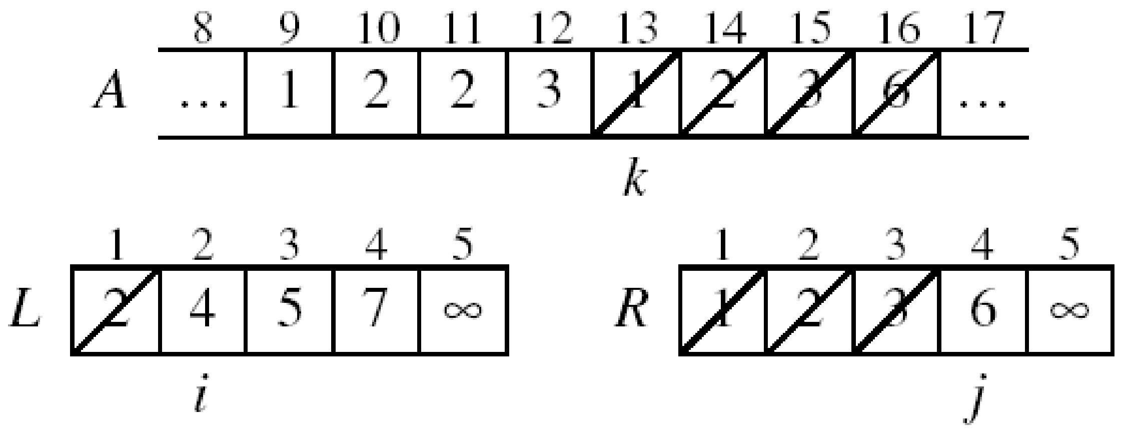
Example: MERGE(A, 9, 12, 16)

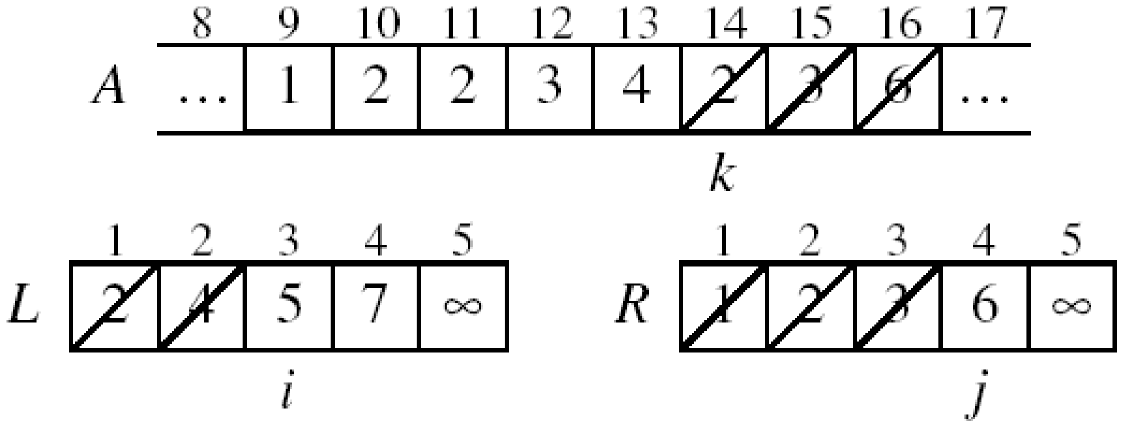


Example: MERGE(A, 9, 12, 16)

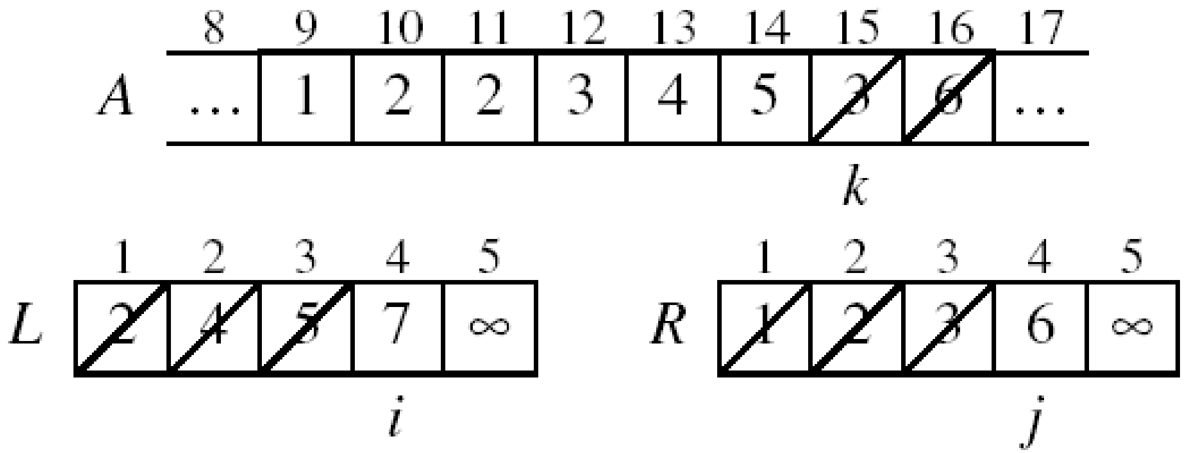


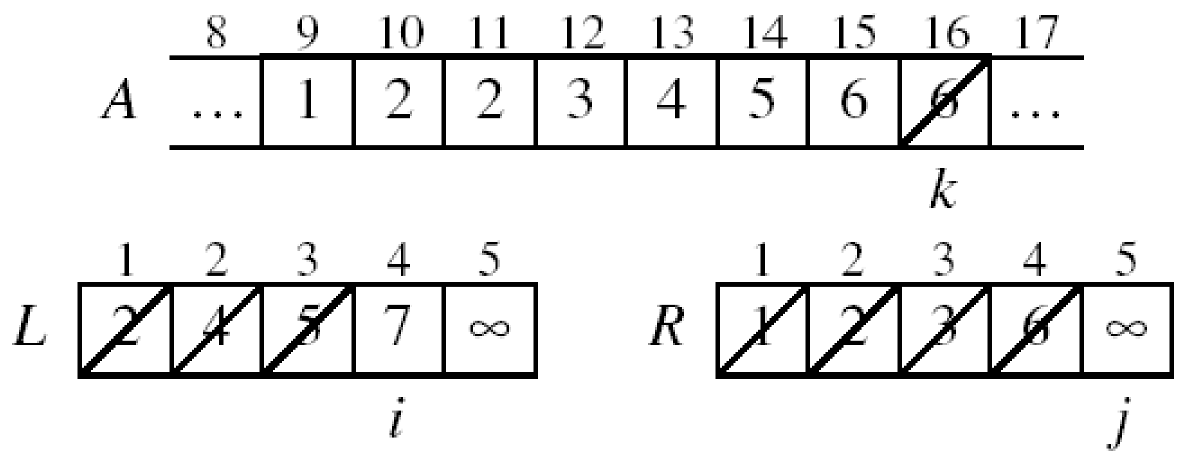
Example (cont.)



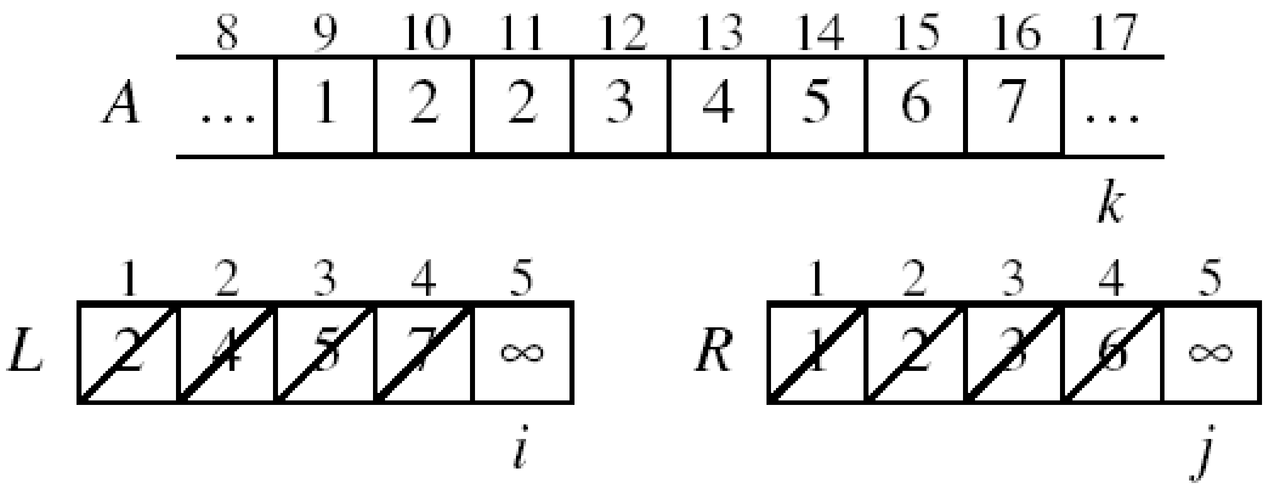


Example (cont.)





Example (cont.)

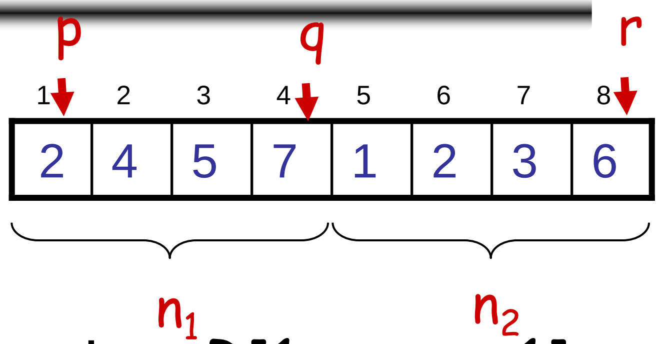


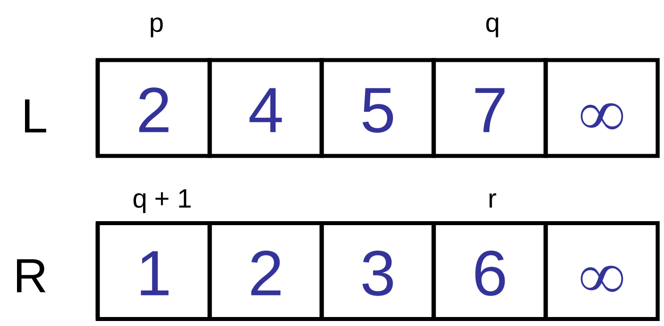
Done!

Merge - Pseudocode

Alg.: MERGE(A, p, q, r)

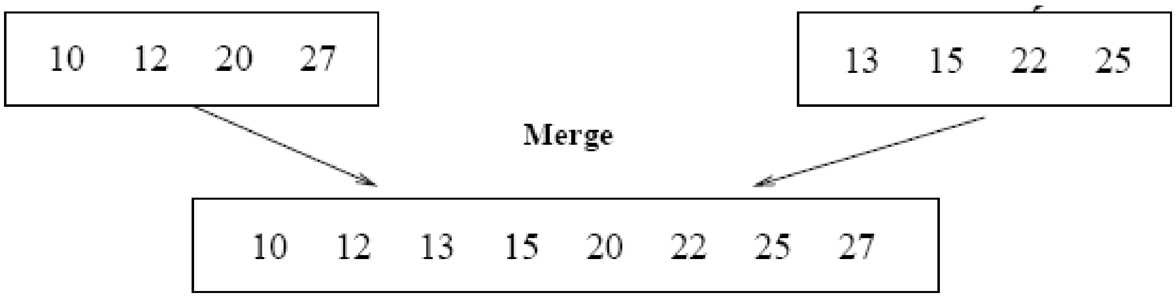
1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \,..\, n_1 + 1]$ and the next n_2 elements into $R[1 \,..\, n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\,i\,] \le R[\,j\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\,j\,]$
10. $j \leftarrow j + 1$





Running Time of Merge (assume last **for** loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



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 $aT(n/b)$
  - **Combine** the solutions $C(n)$

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

MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2$
    $\Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

Merge Sort - Discussion

- Running time insensitive of the input
- Advantages:
  - Guaranteed to run in $\Theta(n \lg n)$
- Disadvantage
  - Requires extra space $\approx N$

Sorting Challenge 1

Problem: Sort a file of huge records with tiny keys

Example application: Reorganize your MP-3 files

Which method to use?

A. merge sort, guaranteed to run in time ~$N \lg N$
B. selection sort
C. bubble sort
D. a custom algorithm for huge records/tiny keys
E. insertion sort

Sorting Files with Huge Records and Small Keys

- Insertion sort or bubble sort?
  - NO, too many exchanges
- Selection sort?
  - YES, it takes linear time for exchanges
- Merge sort or custom method?
  - Probably not: selection sort simpler, does less swaps

Sorting Challenge 2

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. Bubble sort
B. Selection sort
C. Mergesort guaranteed to run in time ~NlgN
D. Insertion sort

Sorting Huge, Randomly - Ordered Files

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Mergesort?
  - YES, it is designed for this problem

Sorting Challenge 3

Problem: sort a file that is already almost in order

Applications:

- Re-sort a huge database after a few changes
- Doublecheck that someone else sorted a file

Which sorting method to use?

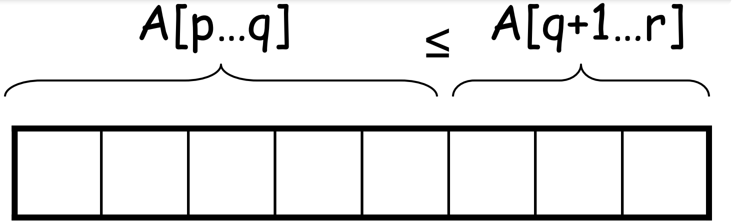
A. Mergesort, guaranteed to run in time ~NlgN
B. Selection sort
C. Bubble sort
D. A custom algorithm for almost in-order files
E. Insertion sort

Sorting Files That are Almost in Order

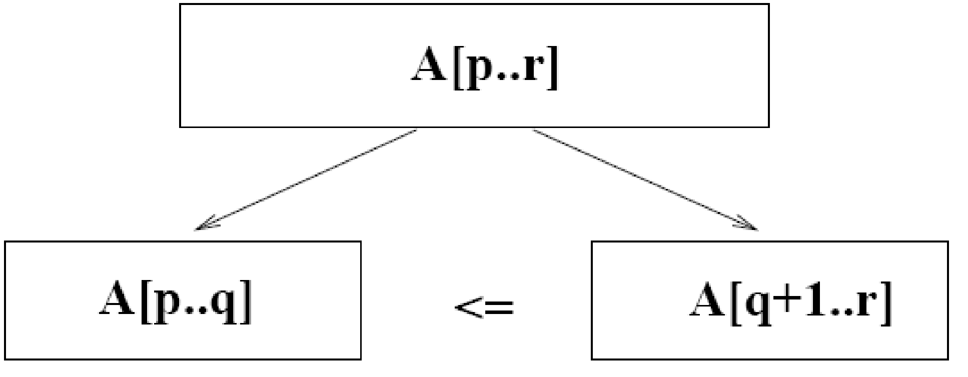
- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, bad for some definitions of “almost in order”
  - Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Insertion sort?
  - YES, takes linear time for most definitions of “almost in order”
- Mergesort or custom method?
  - Probably not: insertion sort simpler and faster

Quicksort

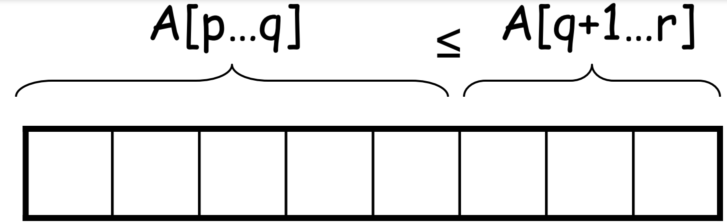
- Sort an array $A[p...r]$



- **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 $A[q+1..r]$
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

QUICKSORT

Alg.: QUICKSORT(A, p, r) Initially: p=1, r=n

if p < r

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

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

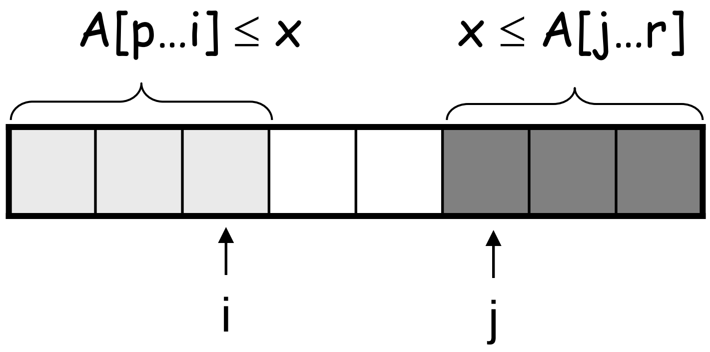
T(n) = T(q) + T(n – q) + f(n) ($f(n)$ depends on PARTITION())

Partitioning the Array

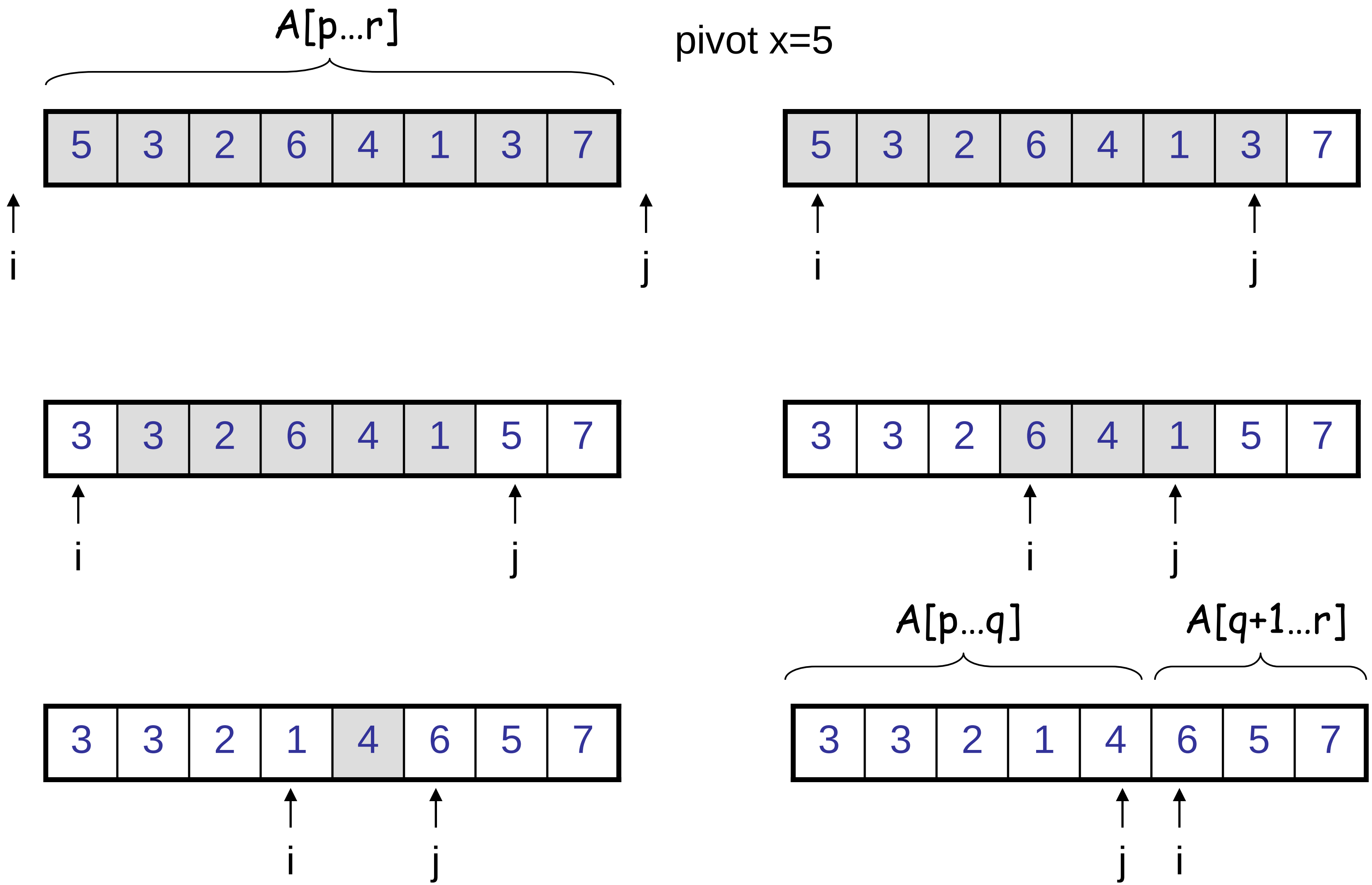
- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition (see prob. 7-1, page 159)
  - Select a pivot element **x** around which to partition
  - Grows two regions

    $A[p \ldots i] \leq \mathbf{x}$

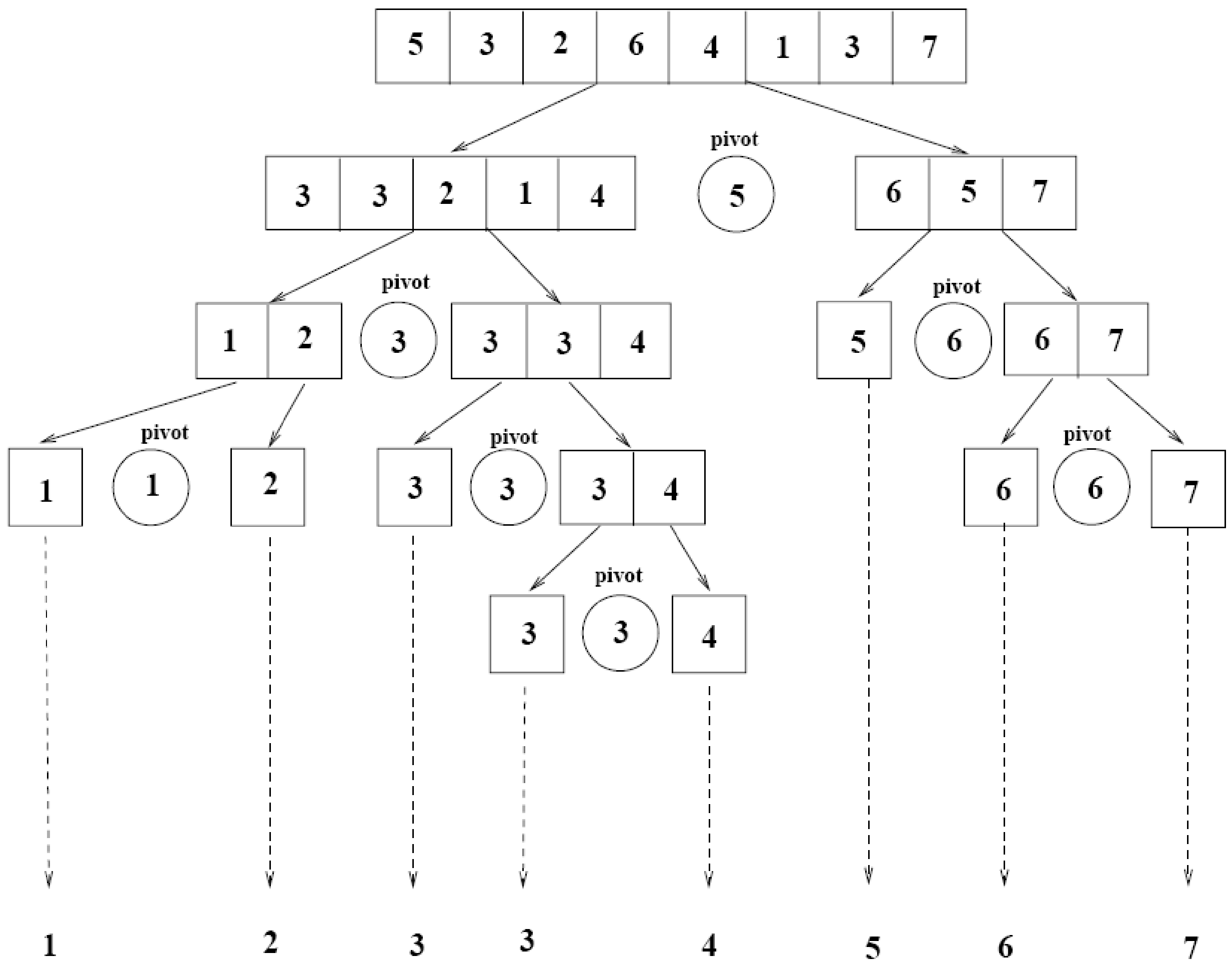
    $\mathbf{x} \leq A[j \ldots r]$



Example


A[p…r]
pivot x=5
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
A[p…q]
A[q+1…r]
3 3 2 1 4 6 5 7
i
j
3 3 2 1 4 6 5 7
j
i

Example

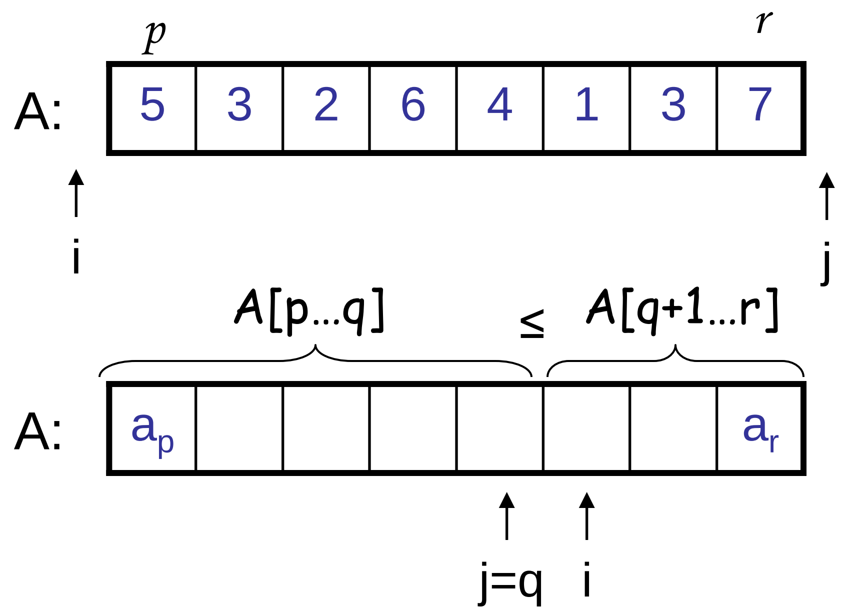


5 3 2 6 4 1 3 7
pivot
5
3 3 2 1 4
6 5 7
pivot
3
1 2
3 3 4
5
pivot
6
6 7
pivot
1
1
2
3
pivot
3
3 4
6
pivot
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

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
        do  repeat i ← i + 1
                until A[i] ≥ x
        if i < j
                then exchange A[i] ↔ A[j]
        else return j
```



Each element is visited once!

Running time: $\Theta(n)$

$n = r - p + 1$

Recurrence

Alg.: QUICKSORT(A, p, r) Initially: p=1, r=n

if $p < r$

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

QUICKSORT (A, p, q)

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

Recurrence:

$$T(n) = T(q) + T(n - q) + n$$

Worst Case Partitioning

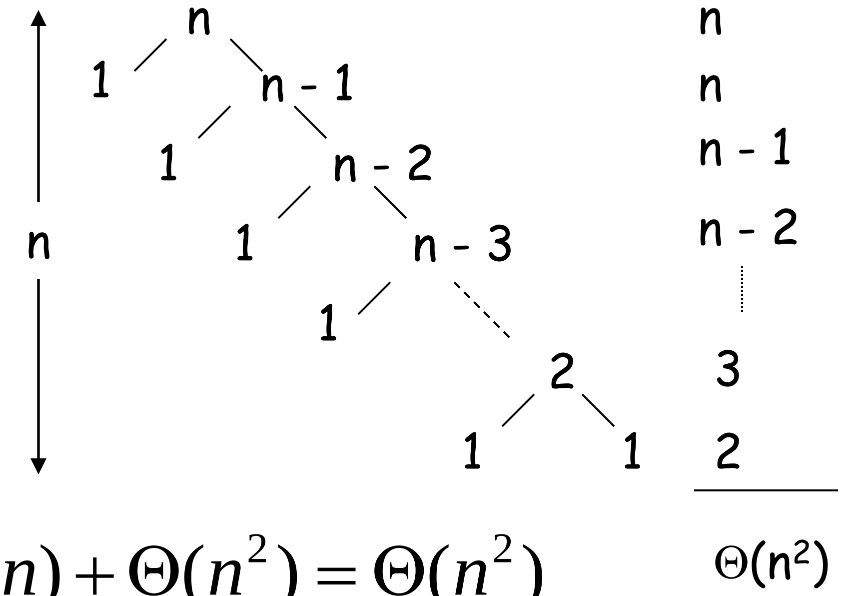
- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

T(n) = T(1) + T(n – 1) + n,

T(1) = Θ(1)

T(n) = T(n – 1) + n

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



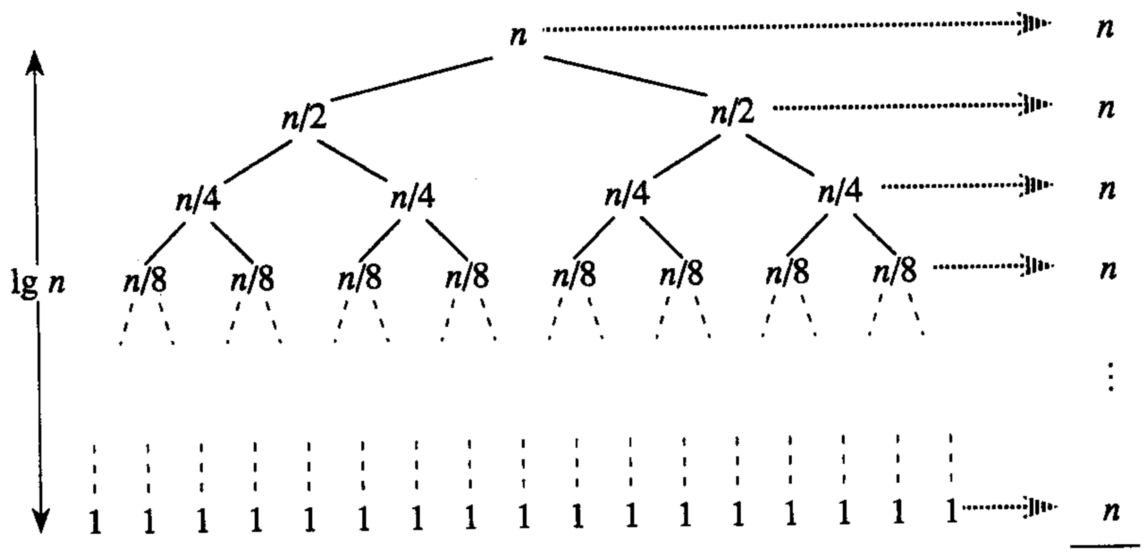
When does the worst case happen?

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2

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

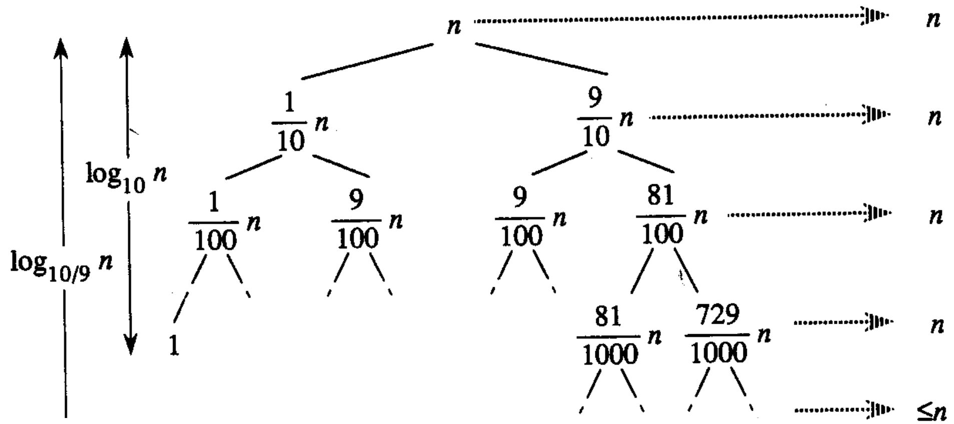
$T(n) = \Theta(n\lg n)$ (Master theorem)



Case Between Worst and Best

- 9-to-1 proportional split

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



- Using the recursion tree:

$$\text{longest path: } Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 n lgn$$

$$\text{shortest path: } Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 n lgn$$

Thus, $Q(n) = \Theta(nlgn)$

How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(nlgn)$ time !!!

- Consider the $(1 : n-1)$ splitting:

ratio=$1/(n-1)$ not a constant !!!

- Consider the $(n/2 : n/2)$ splitting:

ratio=$(n/2)/(n/2) = 1$ it is a constant !!

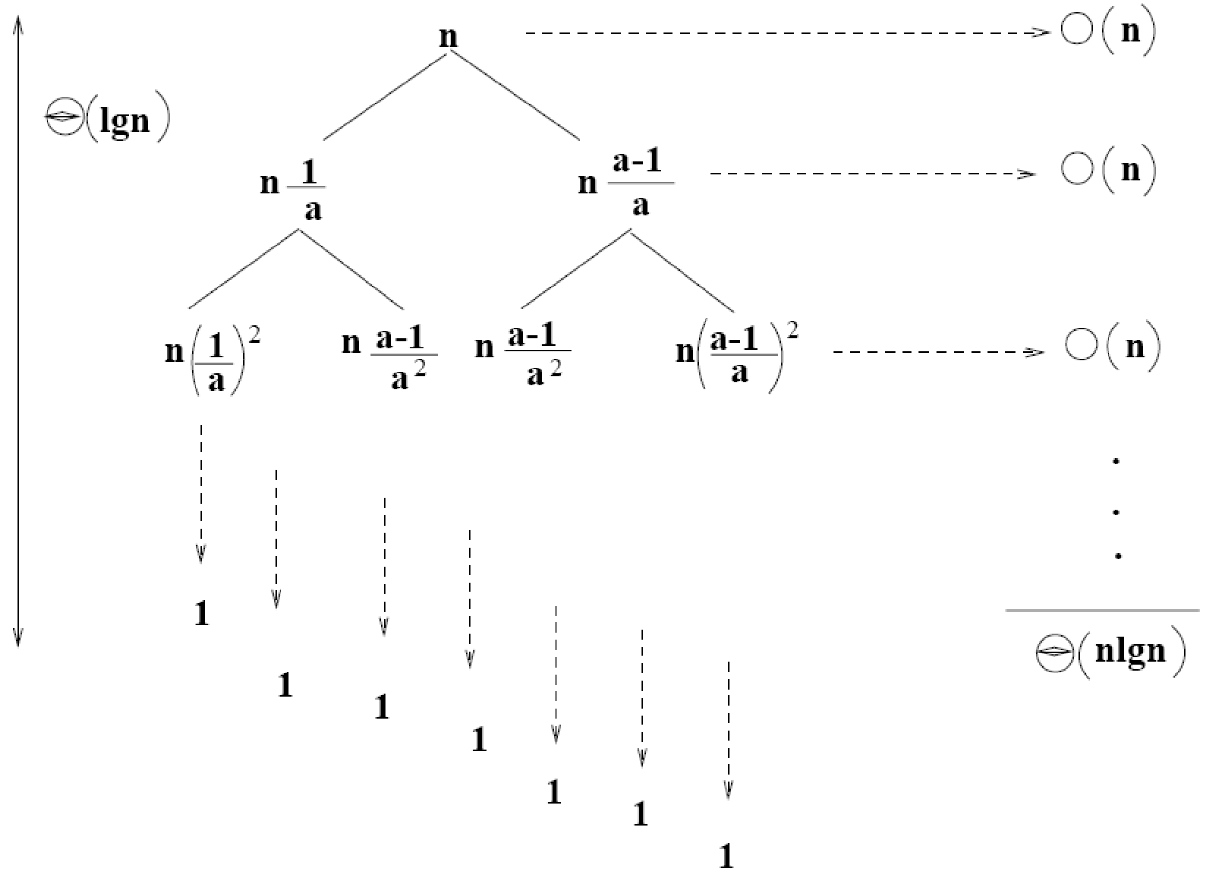
- Consider the $(9n/10 : n/10)$ splitting:

ratio=$(9n/10)/(n/10) = 9$ it is a constant !!

How does partition affect performance?

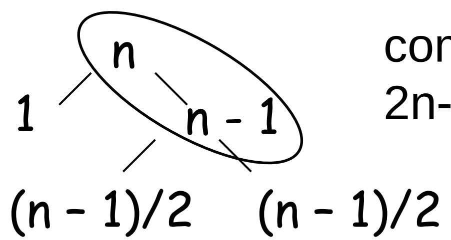
- Any $((a-1)n/a : n/a)$ splitting:

  ratio=$((a-1)n/a)/(n/a) = a-1$ it is a constant !!

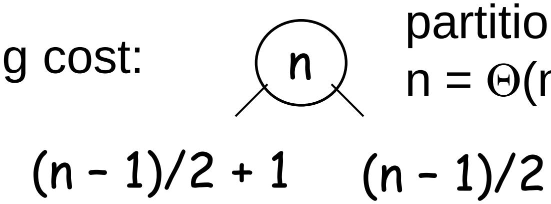


Performance of Quicksort

- 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 across throughout the tree



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



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

Alternate of a good and a bad split

Nearly well balanced split

- Running time of Quicksort when levels alternate between good and bad splits is $O(n\lg n)$