Algorithm Design

(*Divide and Conquer* (*Sorting*) –*Lecture02*)

Presented
By

Dr. Rajesh Purkait
Asst. Professor
Dept. of CSE ITER
(S`O'A Deemed To be University)
E-mail: rajeshpurkait@soa.ac.in

Outline

- Review of previous sorting algorithm
- What is Divide-and-Conquer?
- Merge Sort Approach
  - Describe Merge Sort algorithm
    - Divide
    - Conquer
    - Combine/Merge
- Analyzing Divide-and Conquer Algorithms
- MERGE-SORT Running Time
- Merge Sort – Summery
- Sorting Challenges

Sorting

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

- Bubble Sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

Sorting

- **Selection sort**
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- **Merge Sort**
  - Design approach: divide and conquer
  - Sorts in place: Let's see!!
  - Running time: Let's see!!

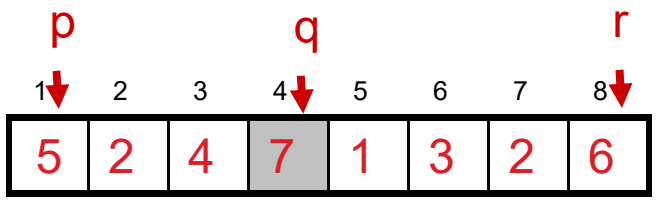
Divide-and-Conquer

- **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 $A[p \ldots r]$:
- **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

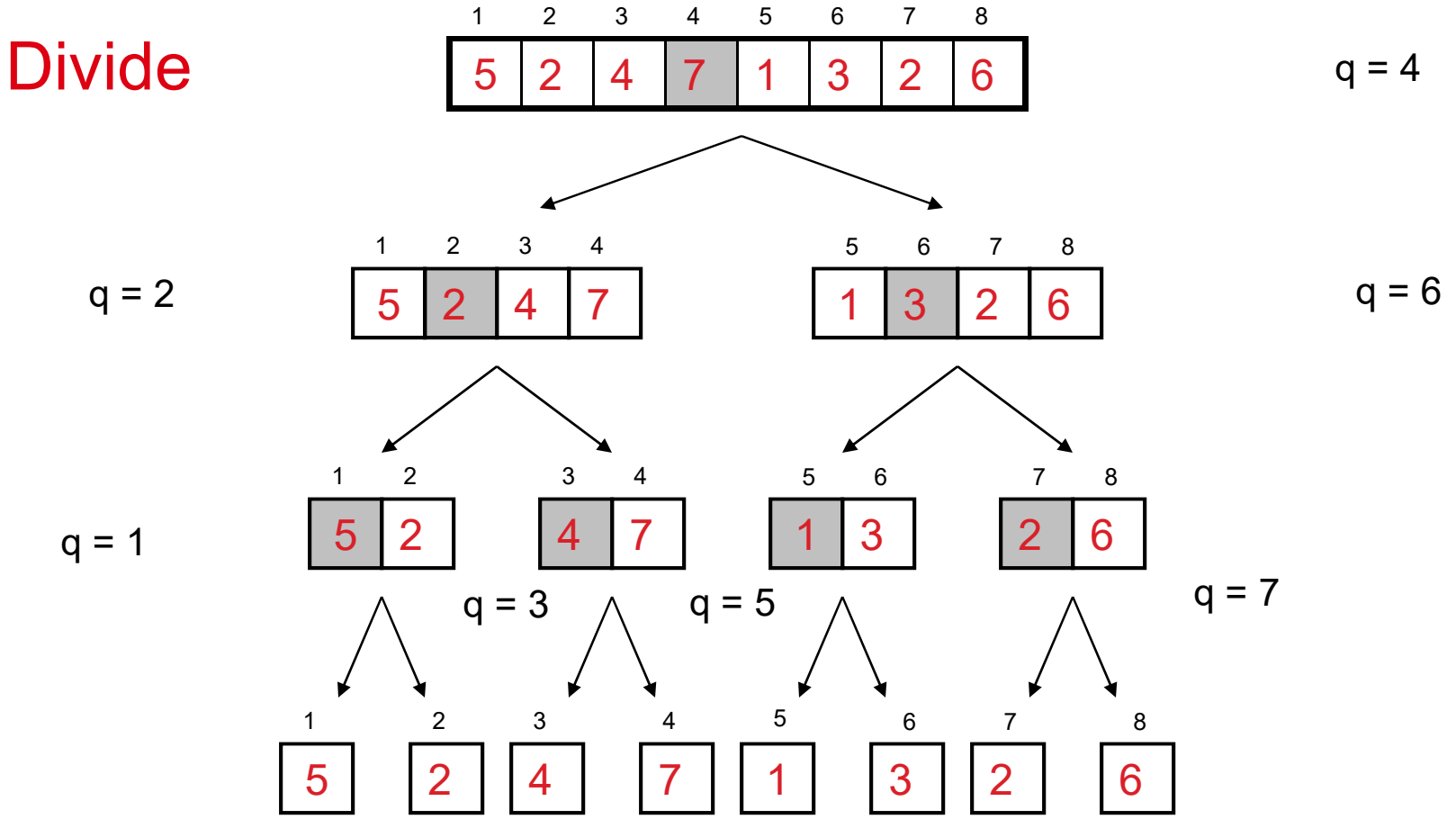


Alg.: MERGE-SORT(A, p, r)

```
if p < r                                  ▷ Check for base case
   then q ← ⌊(p + r)/2⌋                   ▷ Divide
        MERGE-SORT(A, p, q)               ▷ Conquer
        MERGE-SORT(A, q + 1, r)           ▷ Conquer
        MERGE(A, p, q, r)                 ▷ Combine
```

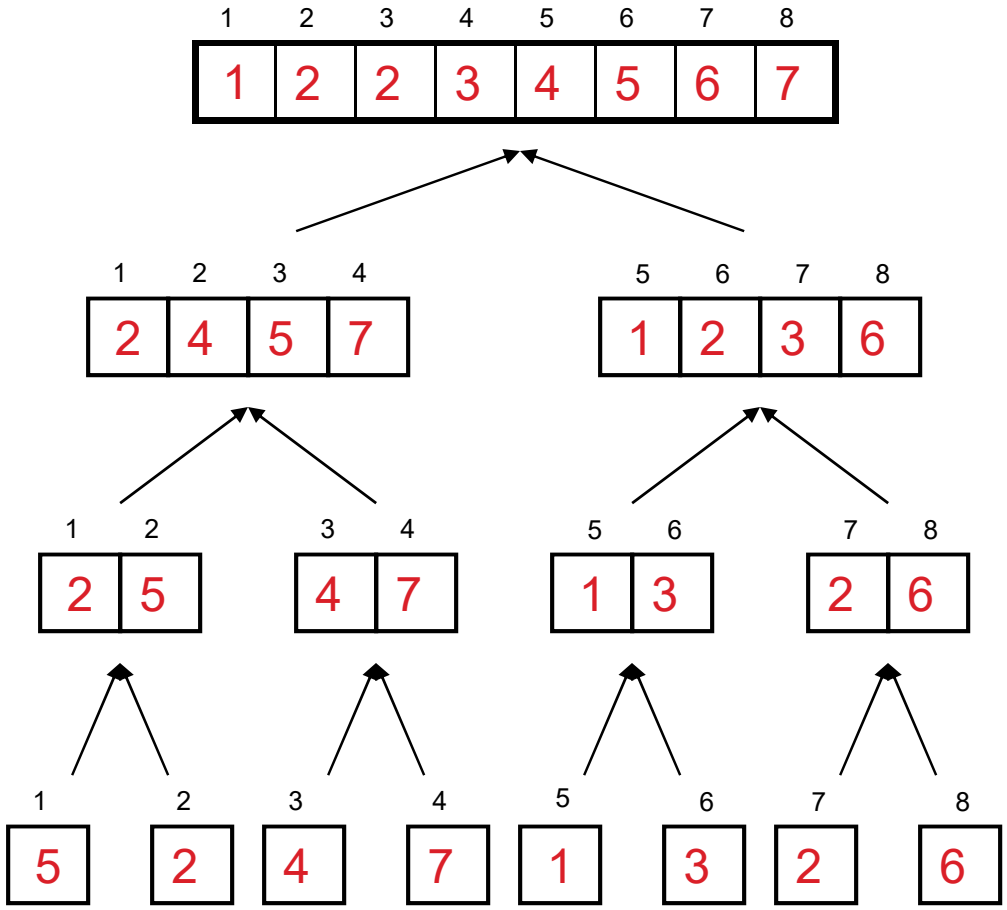
- Initial call: MERGE-SORT(A, 1, n)

Example – n Power of 2

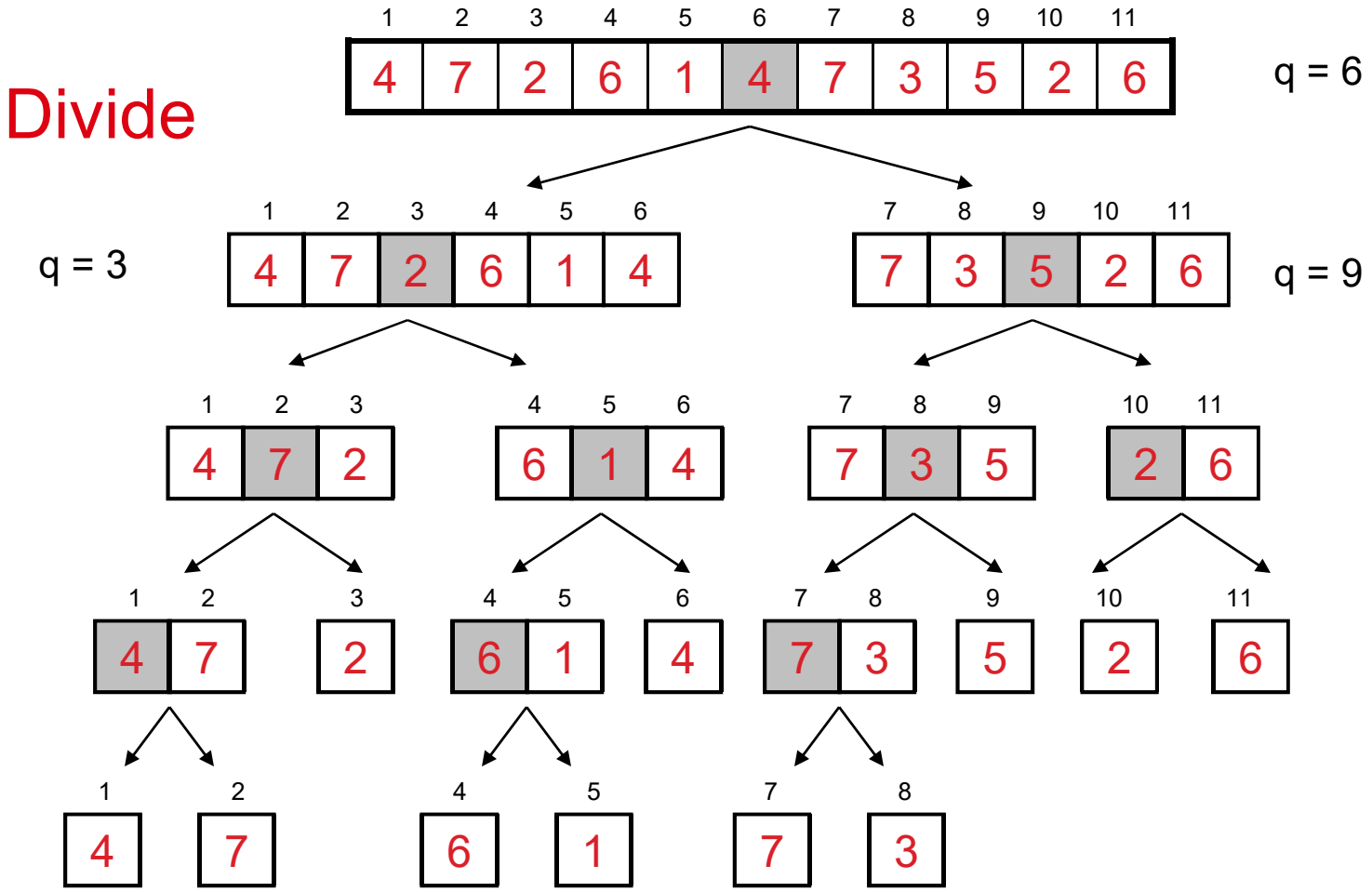


Example – n Power of 2

Conquer and Merge

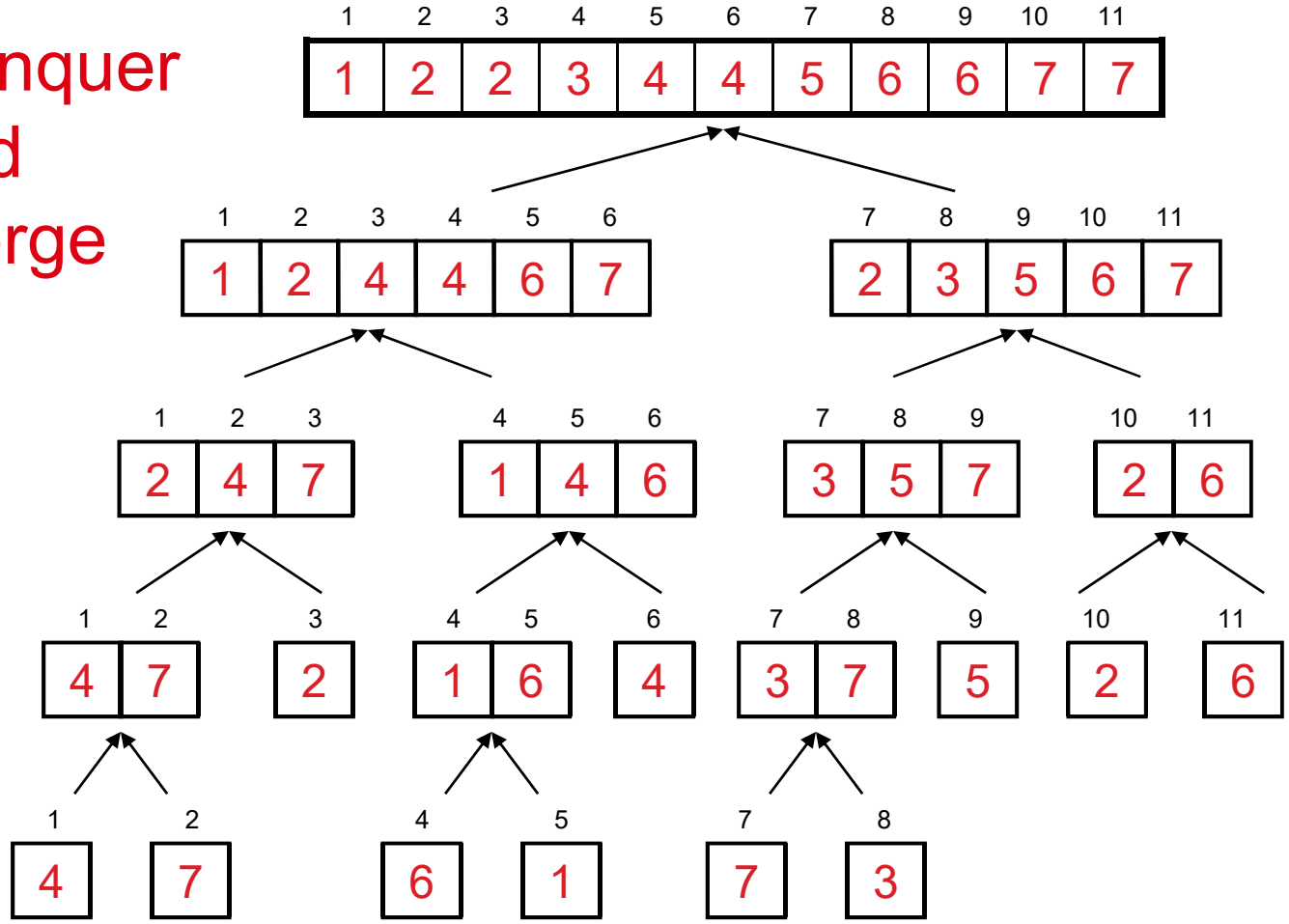


Example2 – n Not a Power of 2

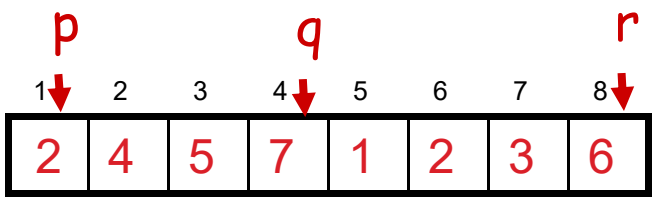


Example – n Not a Power of 2

Conquer and Merge

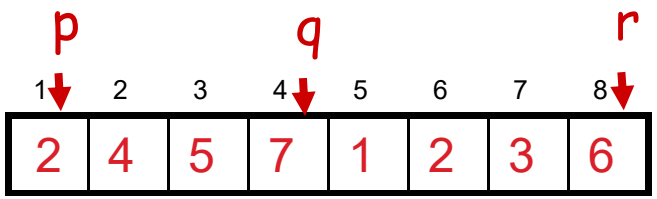


Merging

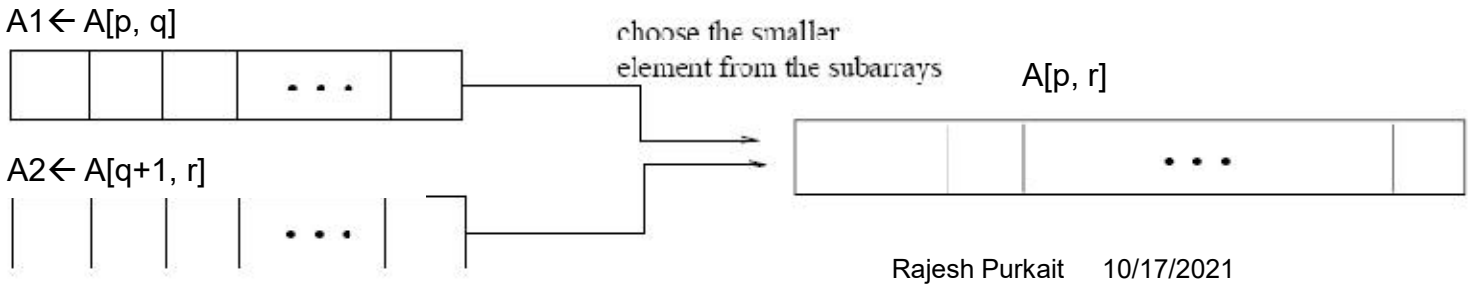


- **Input**: Array A and indices p, q, r such that $p \leq q < r$
  - Subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are sorted
- **Output**: One single sorted subarray $A[p\,..\,r]$

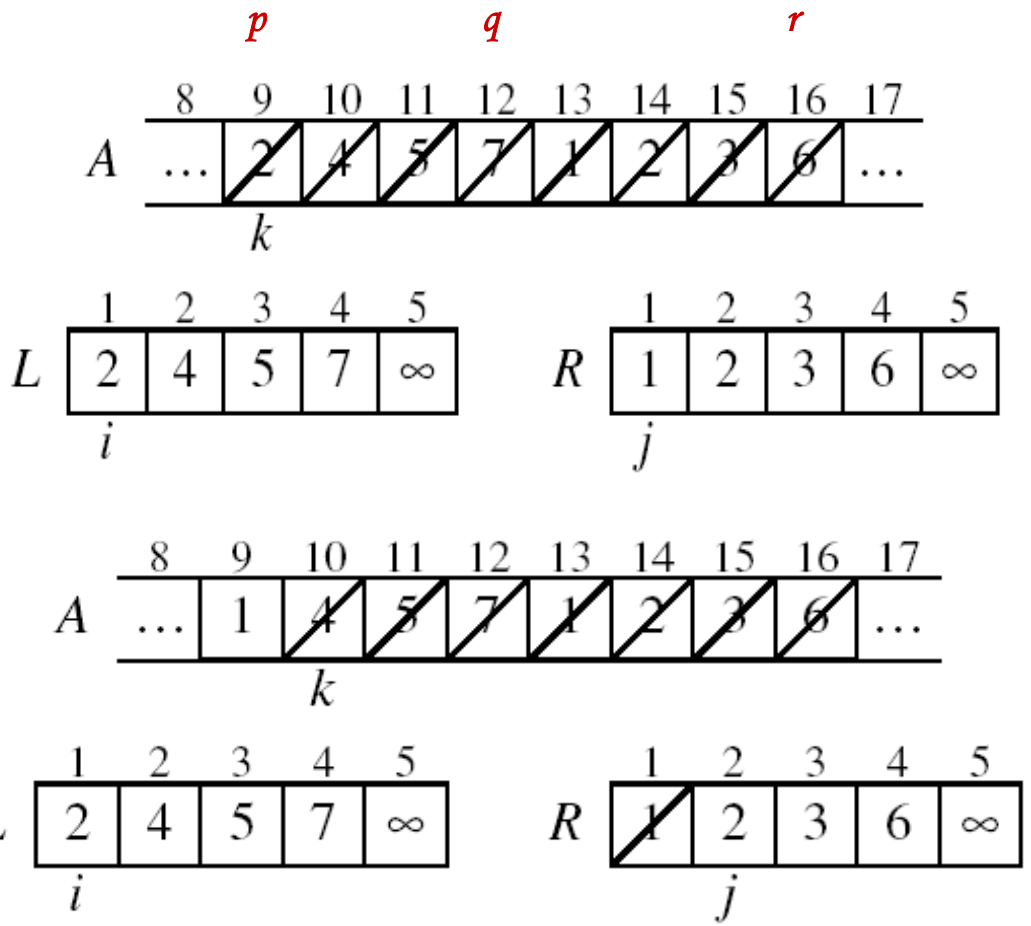
Merging



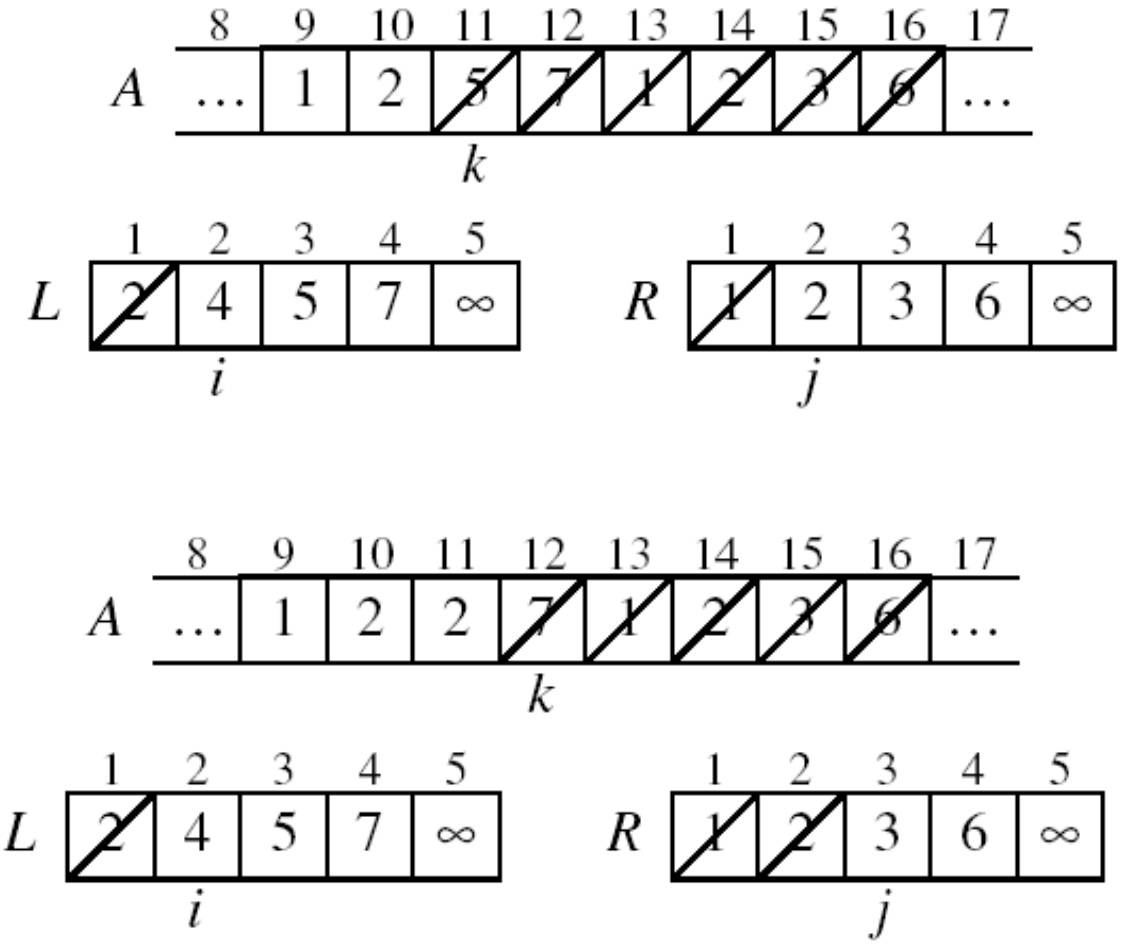
- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile



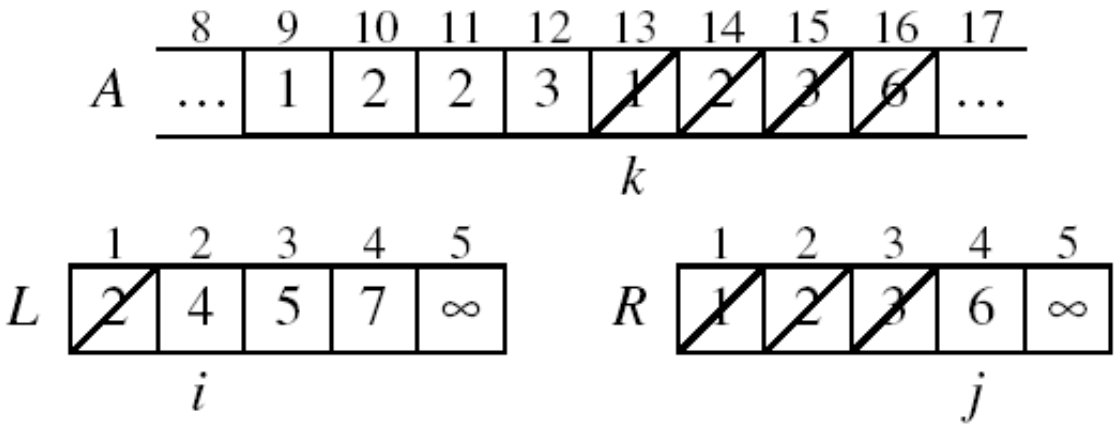
Example: MERGE(A, 9, 12, 16)

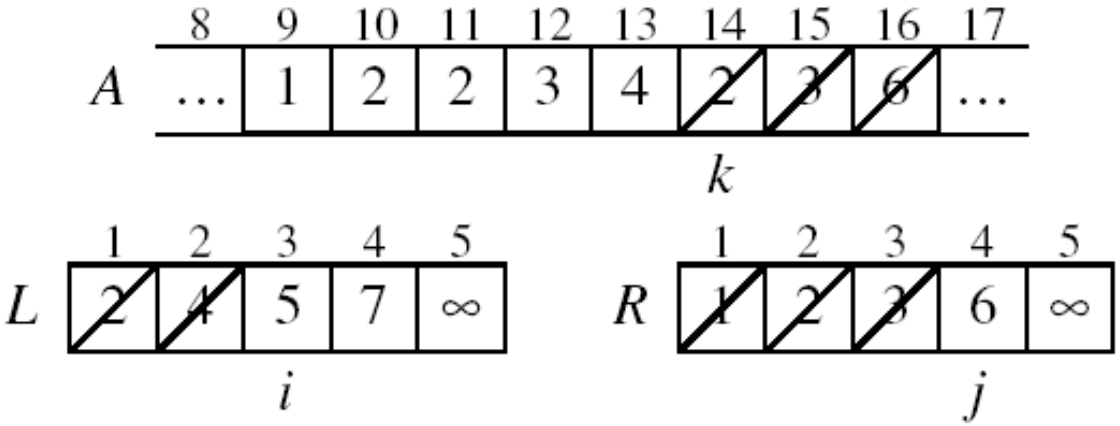


Example: MERGE(A, 9, 12, 16)

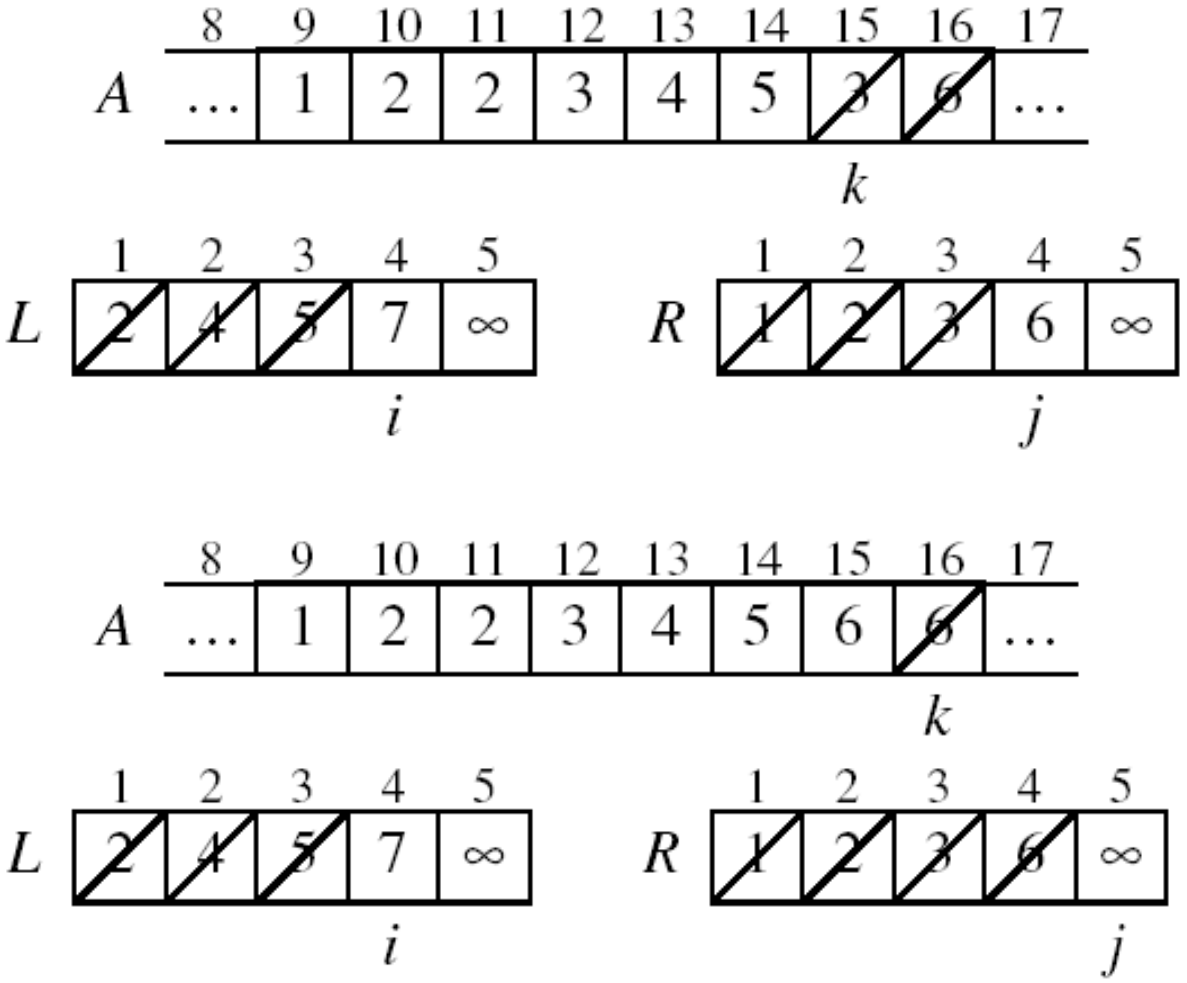


Example (cont.)

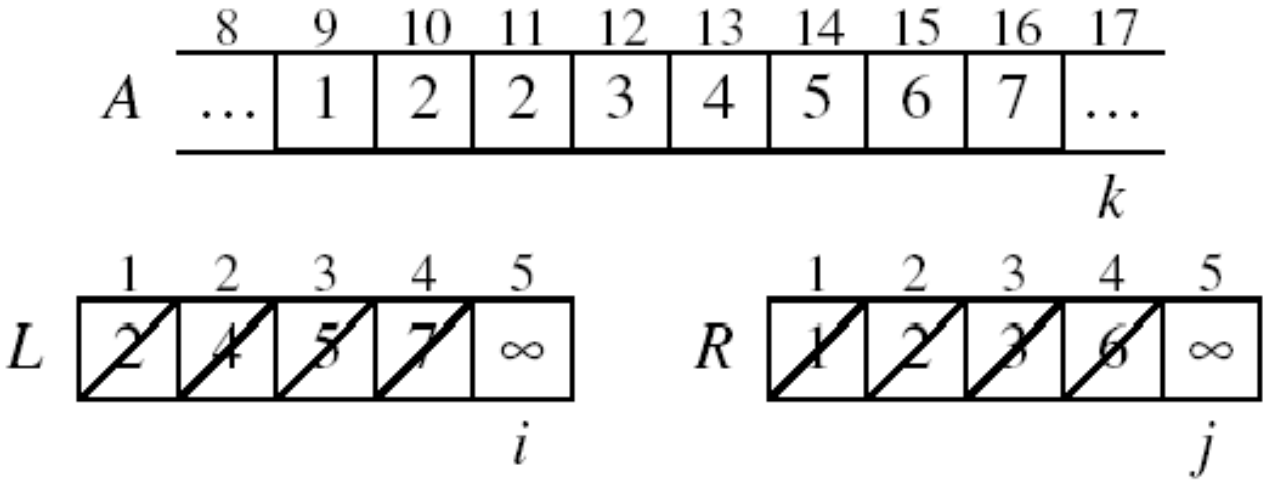




Example (cont.)



Example (cont.)

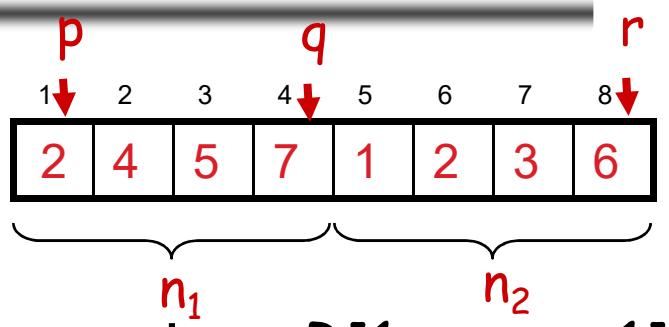


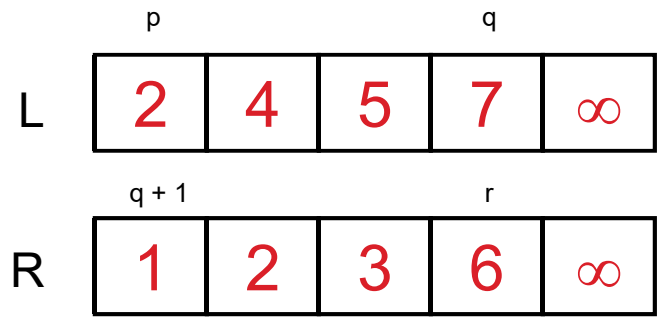
Done!

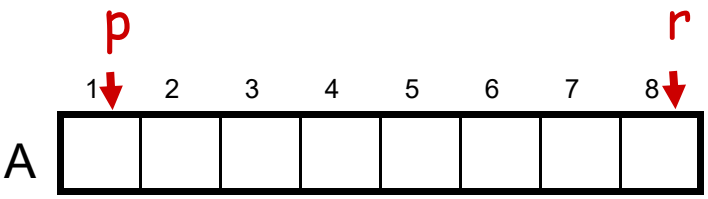
Merge – Pseudocode

Alg1.: 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 $L[\, i \,] \leq R[\, j \,]$
7. then $A[k] \leftarrow L[\, i \,]$
8. $i \leftarrow i + 1$
9. else $A[k] \leftarrow R[\, j \,]$
10. $j \leftarrow j + 1$



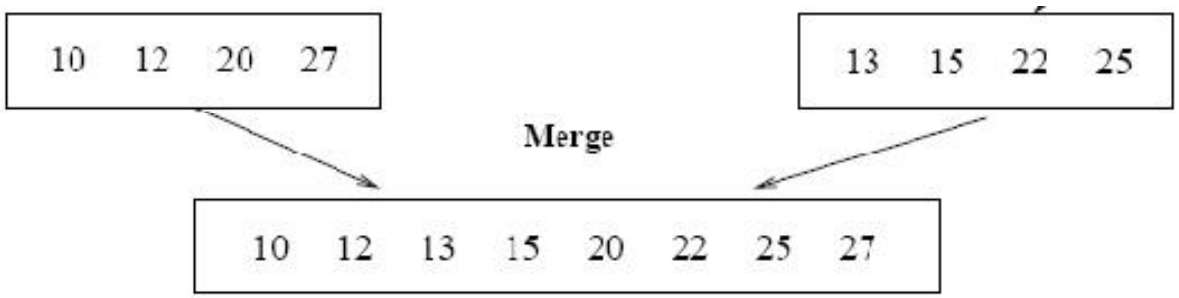




Running Time of Merge(assume last for loop)

- 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** sub-problems, each of size **n/b**: takes D(n)
  - **Conquer** (solve) the sub-problems 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: $T(n) = \Theta(n \lg n)$

Analysis of Merge sort based on number of comparisons

As in the binary search algorithm, the basic operation here is element comparison. That is, the running time is proportional to the number of element comparisons performed by the algorithm. Now, we wish to compute the number of element comparisons $C(n)$ required by Algorithm MERGESORT to sort an array of n elements. For simplicity, we will assume that n is a power of 2, i.e., $n = 2^k$ for some integer $k \geq 0$. If $n = 1$, then the algorithm does not perform any element comparisons. If $n > 1$, then steps 2 through 5 are executed. By definition of the function C, the number of comparisons required to execute steps 3 and 4 is $C(n/2)$ each. By Observation *Alg1*, the number of comparisons needed to merge the two subarrays is between $n/2$ and $n - 1$.

Analysis of Merge sort based on number of comparisons

Thus, the minimum number of comparisons done by the algorithm is given by the recurrence

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2C(n/2) + n/2 & \text{if } n \geq 2. \end{cases}$$

$$C(n) = \frac{n \log n}{2}.$$

The maximum number of comparisons done by the algorithm is given by the recurrence

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2C(n/2) + n - 1 & \text{if } n \geq 2 \end{cases}$$

$$C(n) = n \log n - n + 1.$$

Analysis of Merge sort based on number of comparisons

Observation 1 The total number of element comparisons performed by Algorithm MERGESORT to sort an array of size n, *where* n *is a power of* 2, is between $(n \log n)/2$ and $n \log n - n + 1$.

Proof: We proceed to solve this recurrence by expansion as follows

$$\begin{aligned}
C(n) &= 2C(n/2) + n - 1 \\
&= 2(2C(n/2^2) + n/2 - 1) + n - 1 \\
&= 2^2 C(n/2^2) + n - 2 + n - 1 \\
&= 2^2 C(n/2^2) + 2n - 2 - 1 \\
&\vdots \\
&= 2^k C(n/2^k) + kn - 2^{k-1} - 2^{k-2} - \ldots - 2 - 1 \\
&= 2^k C(1) + kn - \sum_{j=0}^{k-1} 2^j \\
&= 2^k \times 0 + kn - (2^k - 1) \\
&= kn - 2^k + 1 \\
&= n \log n - n + 1.
\end{aligned}$$

As a result, we have the following observation:

Merge Sort – Summery

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

Theorem Algorithm MERGESORT sorts an array of n elements in time $\Theta(n \log n)$ and space $\Theta(n)$.

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 ~N*lg*N
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

Thank You