Merge Sort

Algorithm & Correctness

Design paradigms

- **Paradigm**
  - “In science and philosophy, a paradigm is a distinct set of concepts or thought patterns, including theories, research methods, postulates, and standards for what constitutes legitimate contributions to a field”- Wikipedia

Types of Design Paradigms

- Incremental Approach
- Divide and Conquer
- Greedy approach
- Dynamic Programming

Incremental approach

- **Example: Insertion sort**
  - In the so far sorted subarray, insert a new single element into its proper place, resulting in the new sorted subarray
  - Example:

[....] [.........]

A[1 .. j-1] *A [j .. n]*

Key = A [j]

Divide and Conquer

Our life is frittered away by detail. Simplify, simplify.

— Henry David Thoreau

The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

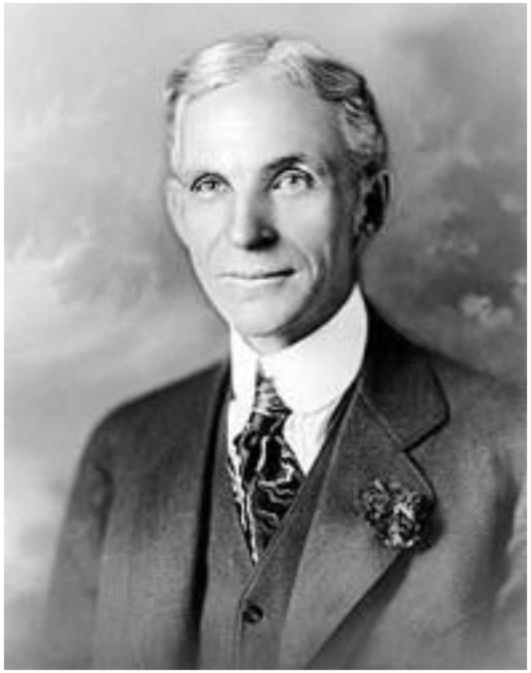
Nothing is particularly hard if you divide it into small jobs.

— Henry Ford

Henry Ford (July 30, 1863 - April 7, 1947) was an American captain of industry and a business magnate.

Founder of the Ford Motor company

Sponsor of the development of the assembly line technique of mass production.



Henry Ford's Assembly line

Divide and Conquer

- **Three crucial steps**
  - **Divide** the problem into smaller sub problems
  - **Conquer** the smaller subproblems recursively.
  - **Combine** solutions of the subproblems to get the solution of the original problem

Divide and conquer - First step

- **Divide/Break** the problem into smaller sub problems
  - For example, Problem P is divided into subproblems P1 and P2.
  - Also, P1 and P2 resemble the original problem and their size is small

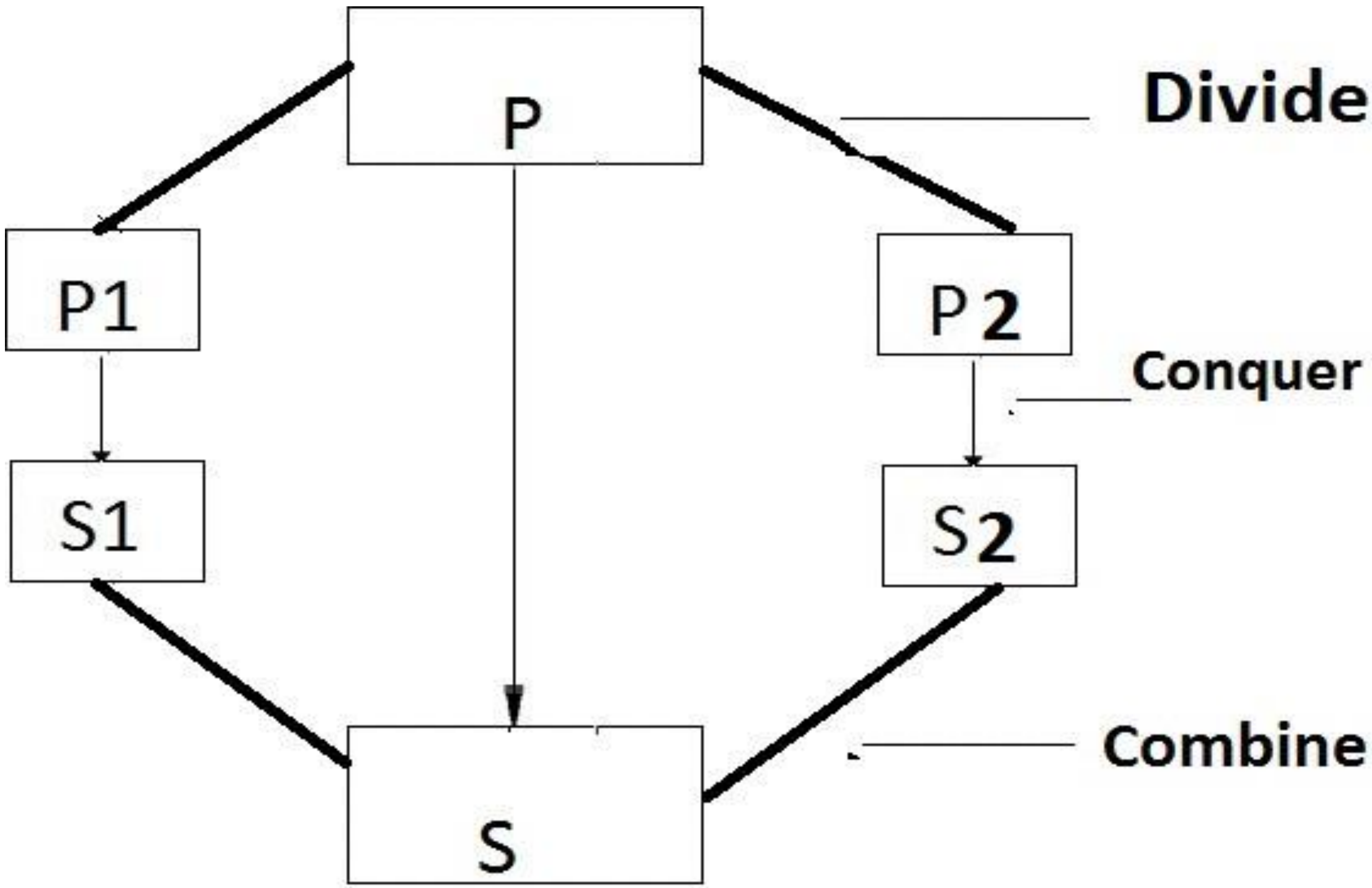
Divide and conquer - Second step

- **Conquer/Solve** the smaller subproblems recursively. If the subproblem size is small, solve them directly
  - P1 is solved to give S1, P2 is solved to give S2

Divide and conquer - Third step

- **Merge/Combine** these solutions to create a solution to the original problem
  - S1 and S2 are combined to give the solution S for the original problem P

Pictorial Representation: Divide and Conquer



Divide and Conquer (D & C)

- Most of the algorithms designed using D & C are recursive in nature
- **Recursive algorithms:** Call themselves recursively to solve the closely related subproblems
- **Examples**
  - Towers of Hanoi
  - Binary search
  - Merge Sort
  - Quick sort

Merge Sort

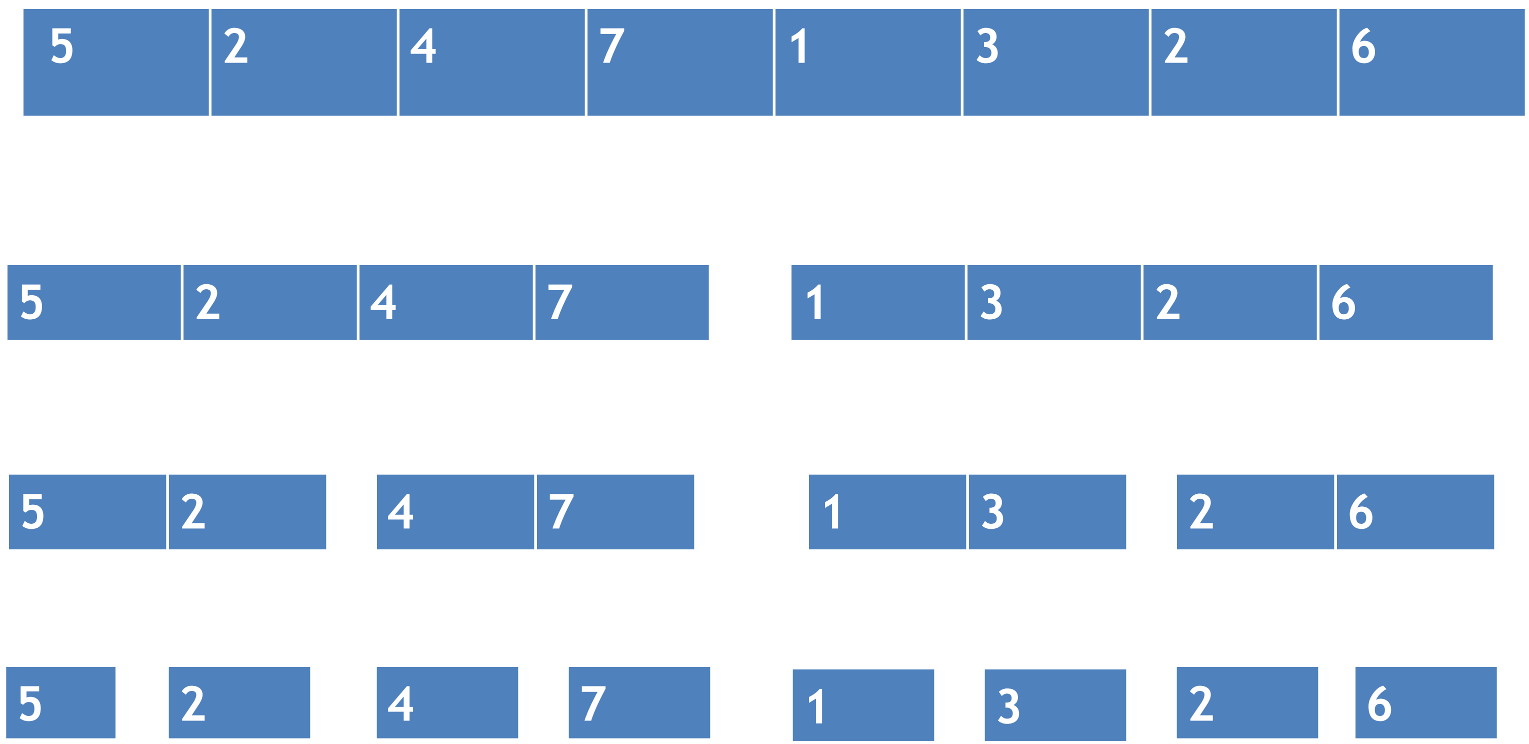
- Follows D & C paradigm

- Divide: Divide the n-element array into two subarrays of size n/2
- Conquer: Sort the two subarrays recursively
- Combine: Merge the two sorted subarrays to produce the sorted array

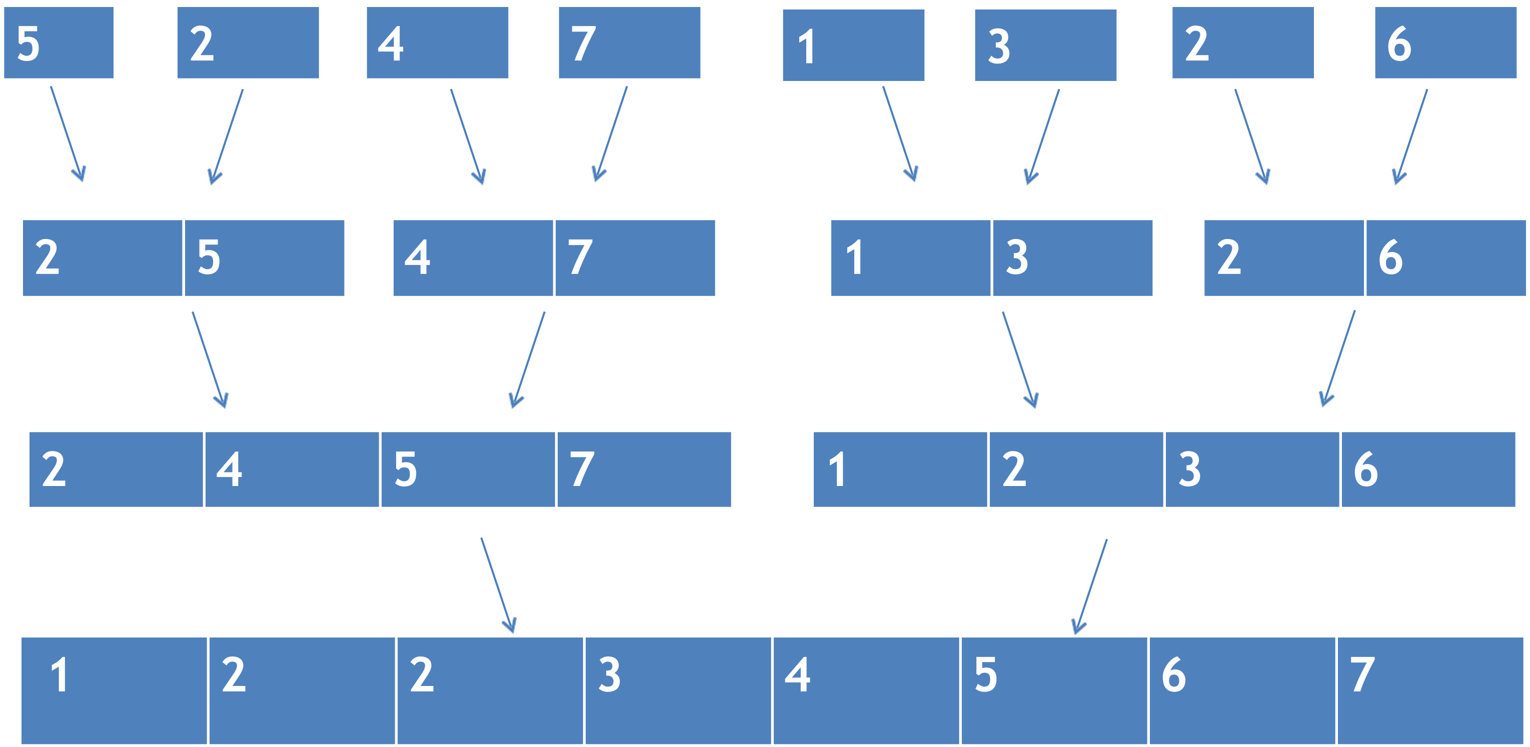
Merge Sort - Example

The operation of merge sort on the array

A= {5, 2, 4, 7, 1, 3, 2, 6}



Merging of sorted subarrays



Merge Sort – Recursive Algorithm

MERGE-SORT(A, p, r)

if $p < r$

  $q = \lfloor (p + r)/2 \rfloor$

  MERGE-SORT(A, p, q)

  MERGE-SORT$(A, q + 1, r)$

  MERGE(A, p, q, r)

Ref: CLRS Book

MERGE-SORT

- If p >= r , the subarray has at most one element and is therefore already sorted.

- Otherwise, the divide step, computes an index q that partitions A[p...r] into two subarrays A[p…q] and A[q+1…r] containing (n/2) elements

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Function call:

MERGE-SORT(A, 1, A.length)

Merge sort - Recursive algorithm

- Base case:
  - When the size of the subproblem is 1, we don’t need to do any further
  - Its already sorted
- Key **operation:** Merging of two sorted arrays in the combine step
- Merge is done by calling another function Merge (A,p,q,r)

Merge function

- Merge is done by calling another function **Merge (A,p,q,r)**

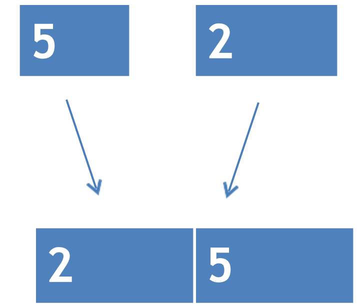
A- Array, p,q,r are indices s.t $p <= q < r$

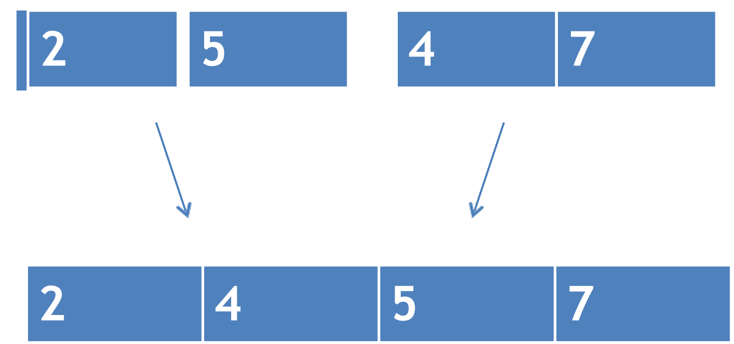
- **Assumption:** A[p…q] and A[q+1… r] are in sorted order
- **Input:** Array A, indices p, q, and r
- **Output:** Merges A[p…q] and A[q+1 … r] and produce a single sorted subarray A[p…r]

Merge function

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

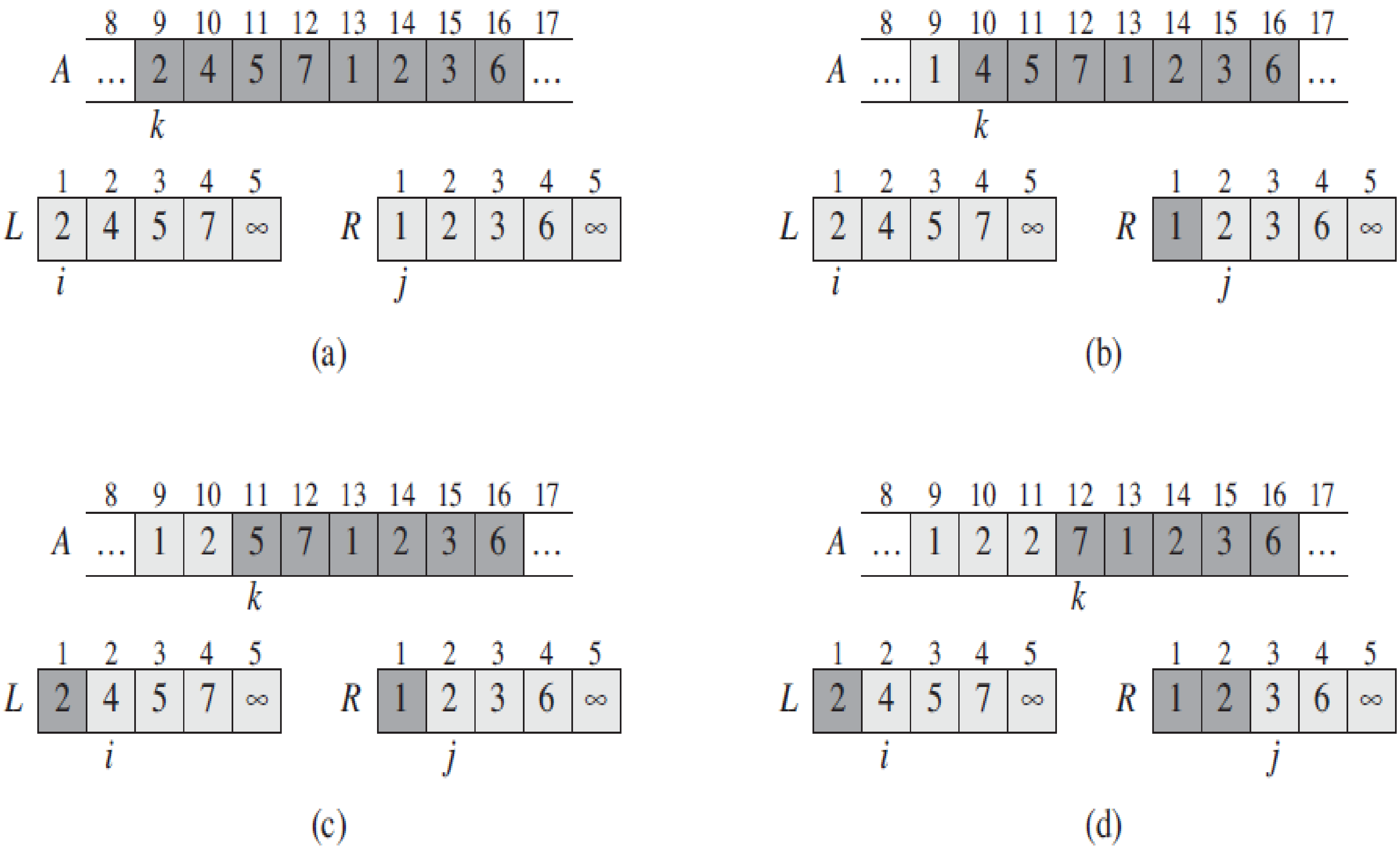
Sentinel - a special value

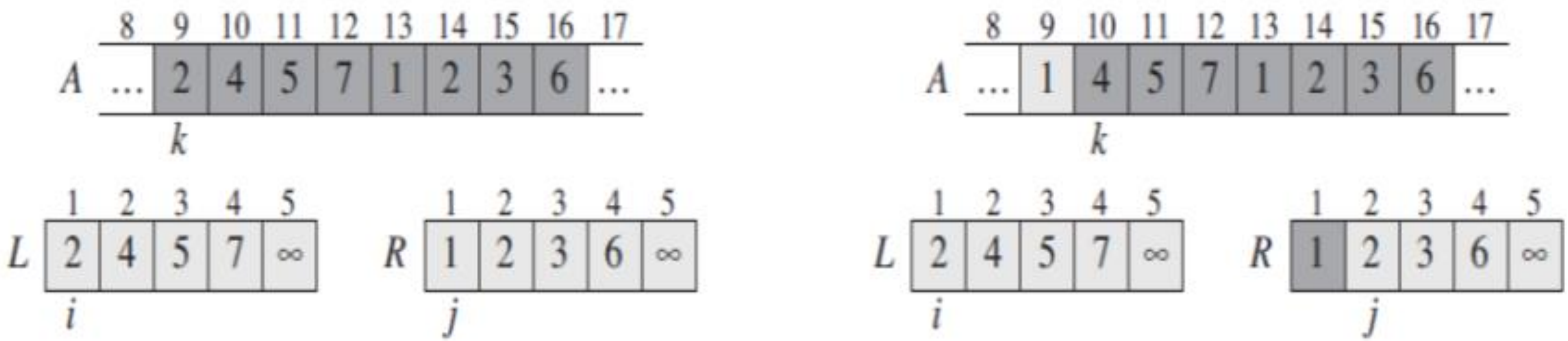




Ref: CLRS Book

Working of Merge function



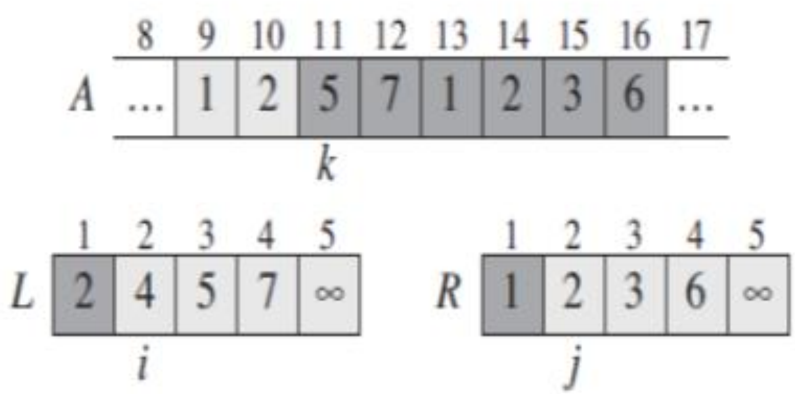


(a) (b)

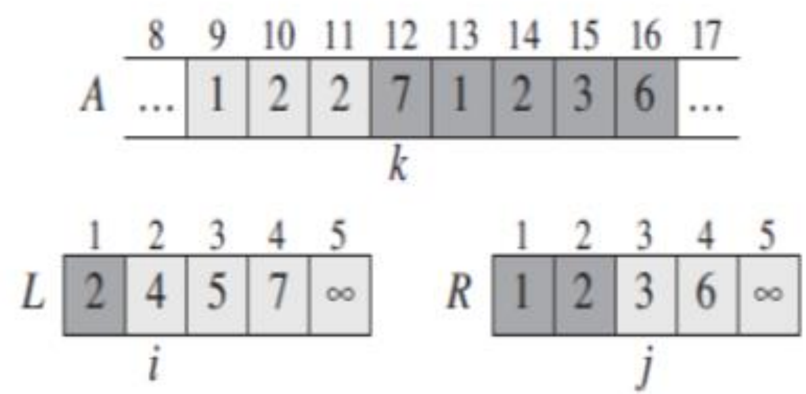
MERGE(A, p, q, r)

```
n₁ = q − p + 1
n₂ = r − q
let L[1 .. n₁ + 1] and R[1 .. n₂ + 1] be new arrays
for i = 1 to n₁
    L[i] = A[p + i − 1]
for j = 1 to n₂
    R[j] = A[q + j]
L[n₁ + 1] = ∞
R[n₂ + 1] = ∞
```

```
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



(c)

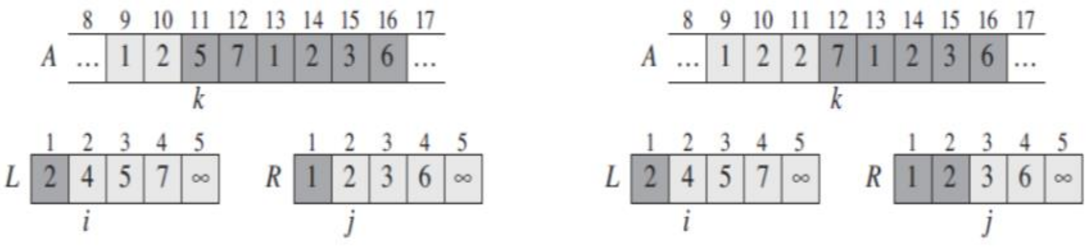


(d)

- Have you understood why do we need to put sentinel values to Left (L) and Right(R) Arrays?

Correctness of Merge

```
12  for k = p to r
13      if L[i] ≤ R[j]
'4          A[k] = L[i]
 5          i = i + 1
 6      else A[k] = R[j]
 7          j = j + 1
```



(c) (d)

- **Loop invariant:** At the start of each iteration of the **for** loop of lines 12–17, the subarray A[p … k-1] contains the $k - p$ smallest elements of $L[1..\ n_1 + 1]$ and $R[1..\ n_2 + 1]$, in sorted order.
- Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

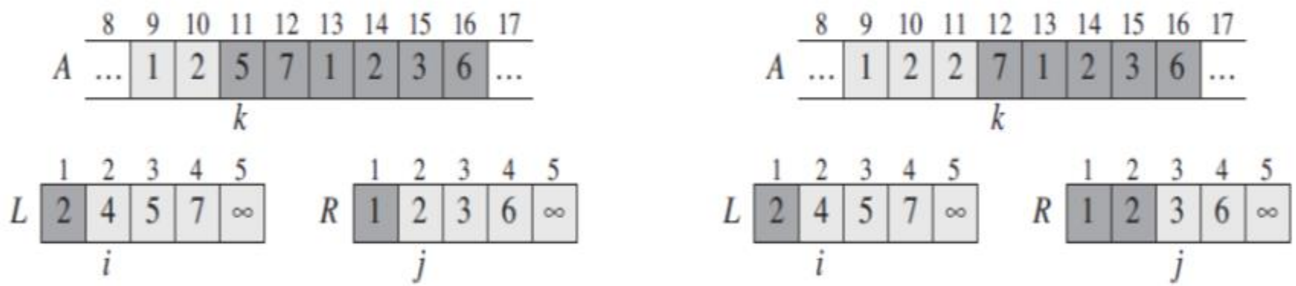
Maintaining the loop invariant

- Show that loop invariant holds **prior to the first iteration** of the *for* loop of lines 12–17
- **Each iteration of the loop** maintains the invariant
- Show correctness when **the loop terminates.**

```
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Initialization

```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

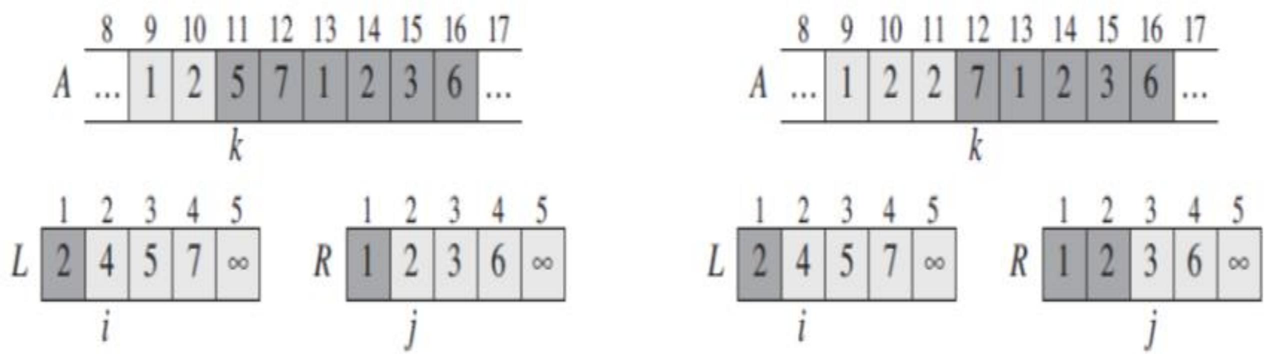


(c) (d)

- Prior to the first iteration of the loop, we have k = p
- Subarray A[p … k- 1] is empty.
- This empty subarray contains the k – p = 0 smallest elements of L and R
- Since i = j = 1, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Maintenance: Each iteration maintains the loop invariant

```
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```



(c) (d)

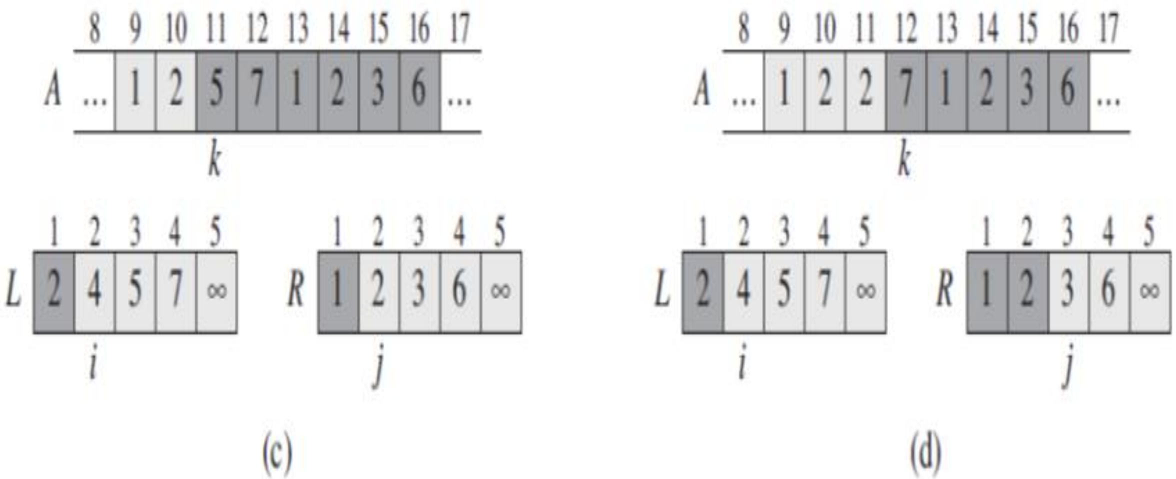
- First, suppose that L[i] <= R[j]
- L[i] is the smallest element not yet copied back into A, because A[p … k- 1] contains k - p smallest elements
- After line 14 copies L[i] into A[k], Subarray A[p … k] will contain k-p+1 smallest elements

```
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

- Incrementing k (in the **for** loop) and i (in line 15) reestablishes the loop invariant for the next iteration
- **Suppose if L[i] > R[j]**, lines 16 – 17 perform appropriate action to maintain loop invariant

Termination

```
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```



- At termination, k = r + 1.
- By the loop invariant, the subarray A[p … k-1] , which is A[p … r], contains k - p smallest elements of L[1.. n_1 + 1] and R[1.. n_2 +1], in sorted order.
- All but the two largest have been copied back into A, and these two largest elements are the sentinels.

Reference: CLRS Book