Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Is sorting in $o(n^2)$ possible?

- **Yes!** but we need to approach the problem differently

Is sorting in $o(n^2)$ possible?

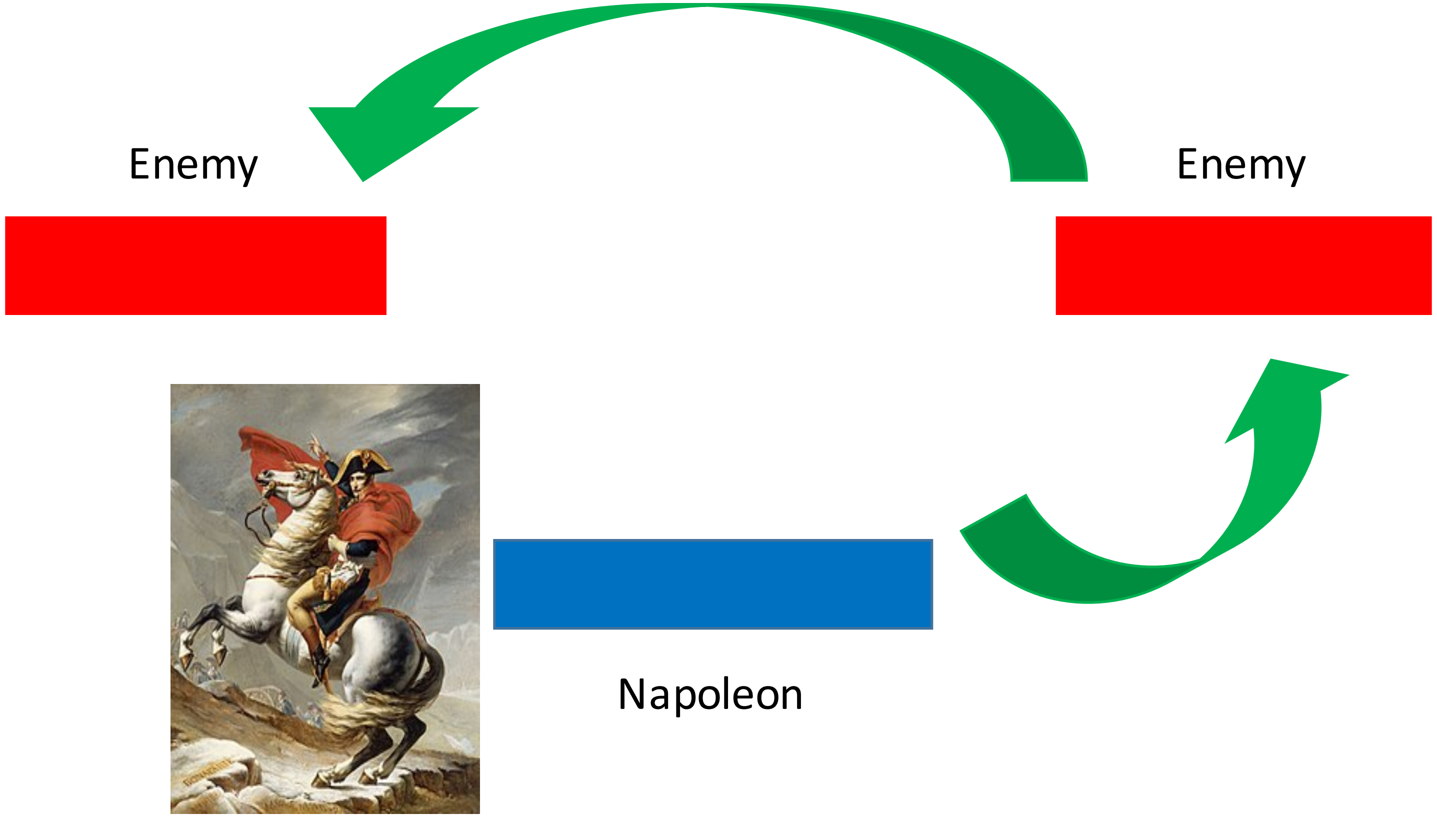
- **Yes!** but we need to approach the problem differently
- So far, we used the **incremental approach** in INSERTION-SORT and SELECTION-SORT, i.e. having a sorted subarray we added new elements to it *one by one*

Divide-and-conquer

- The idea comes from the proverb *“divide et impera”* which means *divide and rule*

Divide-and-conquer

- The idea comes from the proverb *“divide et impera”* which means *divide and rule*
- Napoleon used this strategy



Divide-and-conquer

The **divide-and-conquer paradigm** in algorithm design:

Divide **Conquer** **Combine**

Divide-and-conquer

The **divide-and-conquer paradigm** in algorithm design:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

(Side note on recursion)

Day 1

Empty Kettle

Tea bag

Empty cup

MAKETEA:

1. Fill the kettle with water
2. Boil the water
3. Pour boiled water into the cup
4. Put the tea bag inside the cup
5. Terminate

(Side note on recursion)

Day 2

Kettle filled with water

Tea bag

Empty cup

MakeTea:

~~Fill the kettle with water~~

1. Boil the water
2. Pour boiled water into the cup
3. Put the tea bag inside the cup
4. Terminate

(Side note on recursion)

Recursive approach

Kettle filled with water

Tea bag

Empty cup

MAKETEA:

1. Empty the kettle
2. Solve the problem from day 1

(Side note on recursion)

- Compute $n! = 1 \times 2 \times \cdots \times n$

(Side note on recursion)

- Compute $n! = 1 \times 2 \times \cdots \times n$
- Iterative approach:

```
FACTORIAL-ITERATIVE(n)
    result = 1
    for i = 2 to n
        result = result * i
    return result
```

(Side note on recursion)

- Compute $n! = 1 \times 2 \times \cdots \times n$
- Recursive formula:

$$n! = \underbrace{1 \times 2 \times \cdots \times n - 1}_{(n-1)!} \times n$$

(Side note on recursion)

- Compute $n! = 1 \times 2 \times \cdots \times n$
- Recursive approach:

```
FACTORIAL-RECURSIVE(n)
    if n == 0 or n == 1
        return 1
    return n * FACTORIAL-RECURSIVE(n - 1)
```

(Side note on recursion)

- Compute $n! = 1 \times 2 \times \cdots \times n$
- Recursive approach:

```
FACTORIAL-RECURSIVE(n)
    if n == 0 or n == 1
        return 1
    return n * FACTORIAL-RECURSIVE(n - 1)
```

There is always a ***base case***

There is always a call to the ***same*** *function but* ***with smaller input***

(Side note on recursion)

- Compute $n! = 1 \times 2 \times \cdots \times n$
- Recursive approach:

What is the time complexity?

```
FACTORIAL-RECURSIVE(n)
    if n == 0 or n == 1
        return 1
    return n * FACTORIAL-RECURSIVE(n - 1)
```

There is always a ***base case***

There is always a call to the ***same*** *function but* ***with smaller input***

Time complexity

```
FACTORIAL-RECURSIVE(n)                    T(n)
    if n == 0 or n == 1
        return 1
    return n * FACTORIAL-RECURSIVE(n - 1)
```

Time complexity

```
FACTORIAL-RECURSIVE(n)
    if n == 0 or n == 1
        return 1
    return n * FACTORIAL-RECURSIVE(n - 1)
```

$T(n)$

$O(1)$

Time complexity

```
FACTORIAL-RECURSIVE(n)
    if n == 0 or n == 1
        return 1
    return n * FACTORIAL-RECURSIVE(n - 1)
```

$T(n)$

$O(1)$

Question: How can I describe the running time of the recursive part?

Time complexity

FACTORIAL-RECURSIVE(n) ← $T(n)$

1 **if** $n == 0$ **or** $n == 1$ ← $O(1)$

2 **return** 1

3 **return** n * FACTORIAL-RECURSIVE(n - 1)

Question: How can I describe the running time of the recursive part?

Answer: $T(n-1)$. If solving a problem of size n takes $T(n)$ time, then solving a problem of size $n-1$ takes $T(n-1)$ time.

Time complexity

FACTORIAL-RECURSIVE(n) $T(n)$

if $n == 0$ **or** $n == 1$ $O(1)$

  return 1

return n * FACTORIAL-RECURSIVE(n - 1)

$T(n-1)$

Time complexity

```
FACTORIAL-RECURSIVE(n)
    if n == 0 or n == 1
        return 1
    return n * FACTORIAL-RECURSIVE(n - 1)
```

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ T(n-1) + O(1) & n > 1 \end{cases}$$

Time complexity

- **Comparisons**, the **multiplication (*)**, **calling** another function, and **return** are all simple operations and take constant time, i.e. $O(1)$.
- Note that calling '$\textsc{Factorial-recursive}(n-1)$' is itself a simple operation but the operations that execute as a result of that call are not simple, so we specify its running time using $T(n-1)$
- Later we will talk about how to compute the running time if we have a recursive form for it.

Time complexity

- Assuming $n > 1$, we have

$$T(n) = T(n-1) + O(1)$$

Time complexity

- Assuming $n > 1$, we have

$$T(n) = T(n-1) + O(1)$$

$$= \overbrace{T(n-2) + O(1)} + O(1)$$

Time complexity

- Assuming $n > 1$, we have

$$
\begin{aligned}
T(n) &= T(n-1) + O(1) \\
&= \overbrace{T(n-2) + O(1)} + O(1) \\
&= \overbrace{T(n-3) + O(1)} + O(1) + O(1)
\end{aligned}
$$

Time complexity

- Assuming $n > 1$, we have

$$T(n) = T(n-1) + O(1)$$

$$= \overbrace{T(n-2) + O(1)} + O(1)$$

$$= \overbrace{T(n-3) + O(1)} + O(1) + O(1)$$

$$\vdots$$

$$= \overbrace{O(1) + \cdots + O(1) + O(1)}^{n\ times} = O(n)$$

Time complexity

- Assuming $n > 1$, we have

$$T(n) = T(n-1) + O(1)$$

$$= \overbrace{T(n-2) + O(1)} + O(1)$$

$$= \overbrace{T(n-3) + O(1)} + O(1) + O(1)$$

$$\vdots$$

$$= \overbrace{O(1) + \cdots + O(1) + O(1)}^{n\ times} = O(n)$$

Back to divide-and-conquer

The **divide-and-conquer paradigm** in algorithm design:

Divide the problem into subproblems.

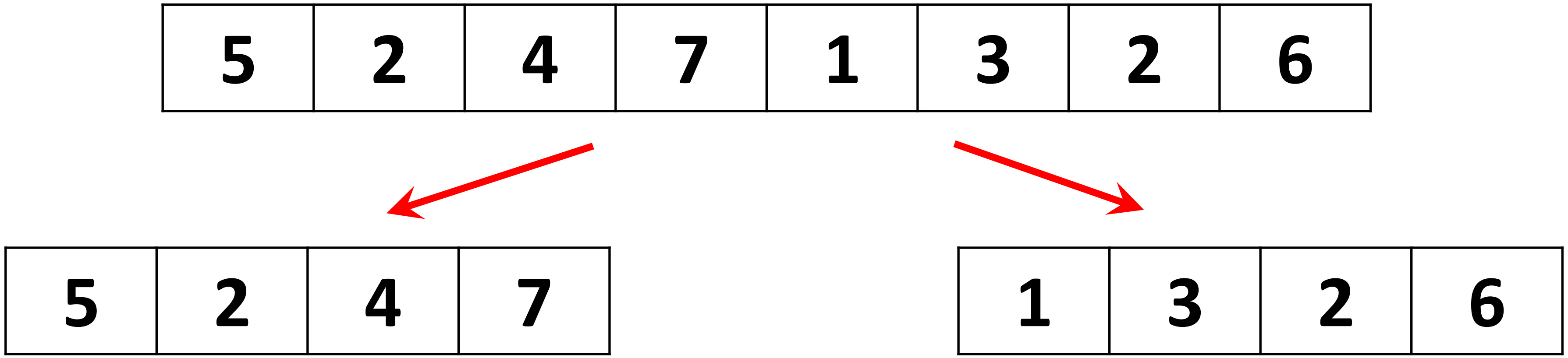
Conquer the subproblems by solving them recursively.

Combine the solutions to the subproblems.

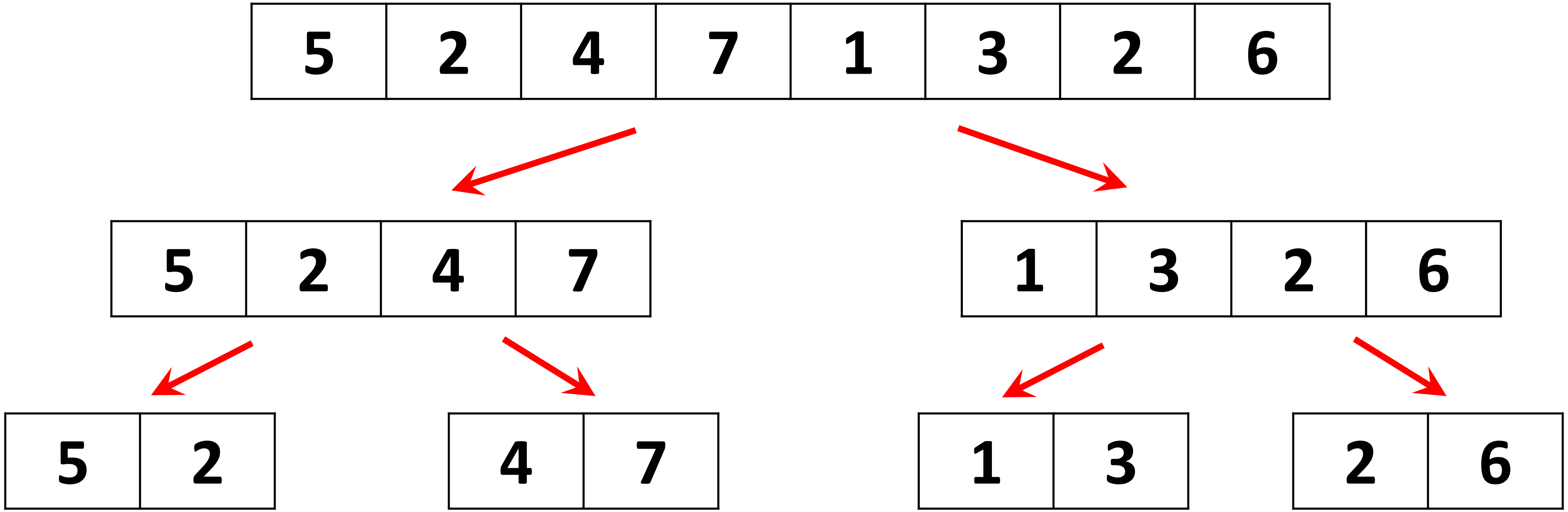
Merge-sort algorithm

5	**2**	**4**	**7**	**1**	**3**	**2**	**6**

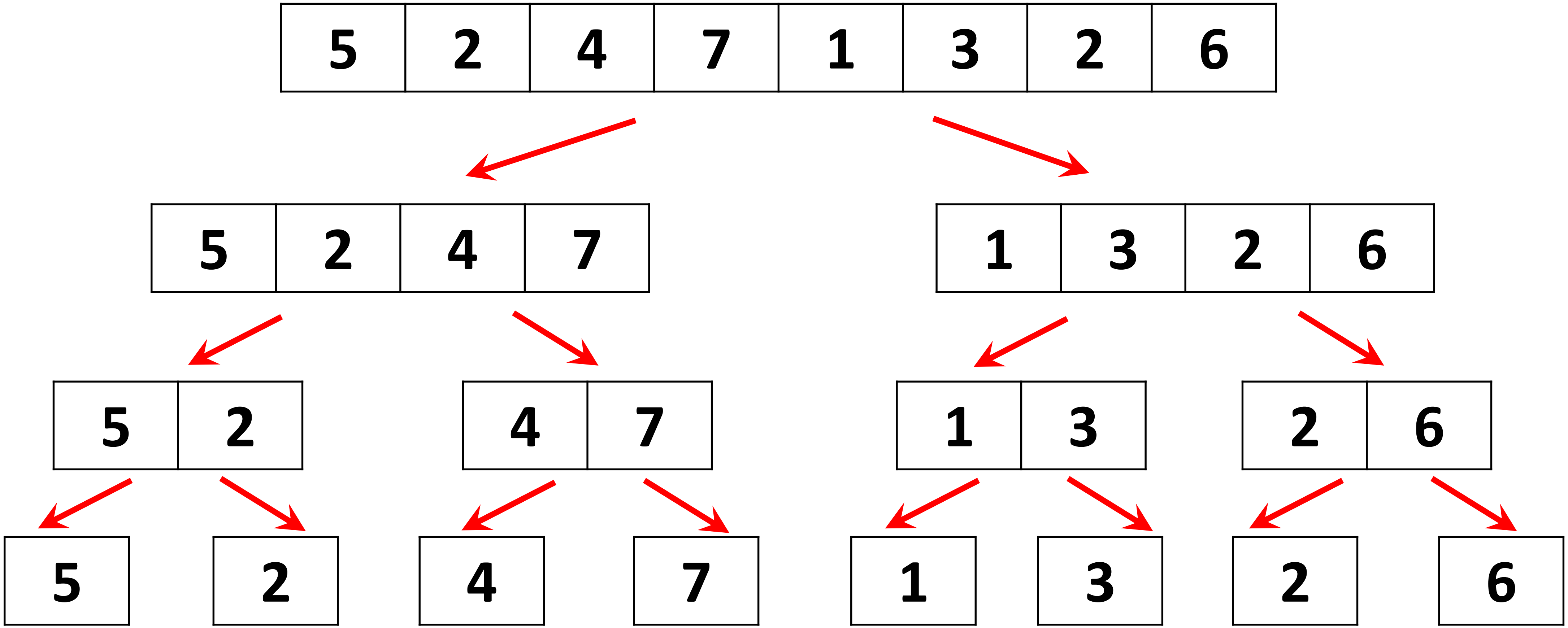
Merge-sort algorithm



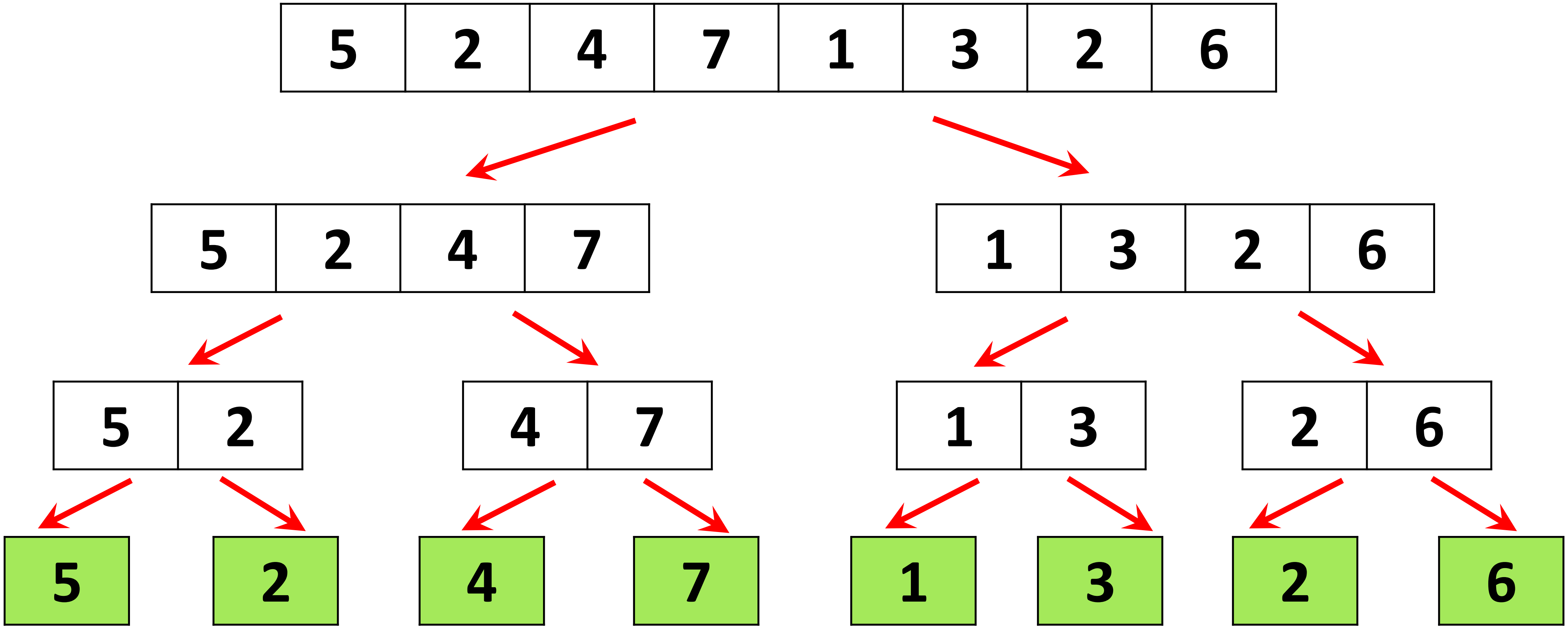
Merge-sort algorithm



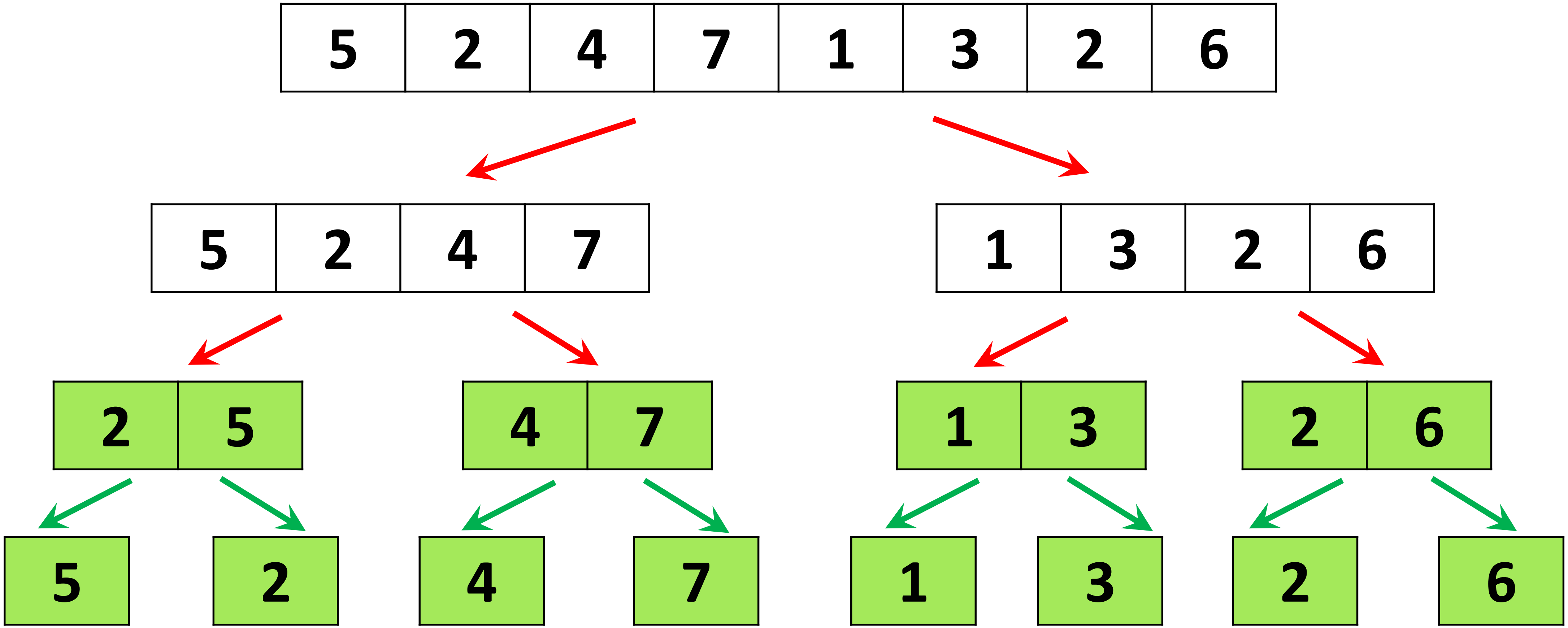
Merge-sort algorithm



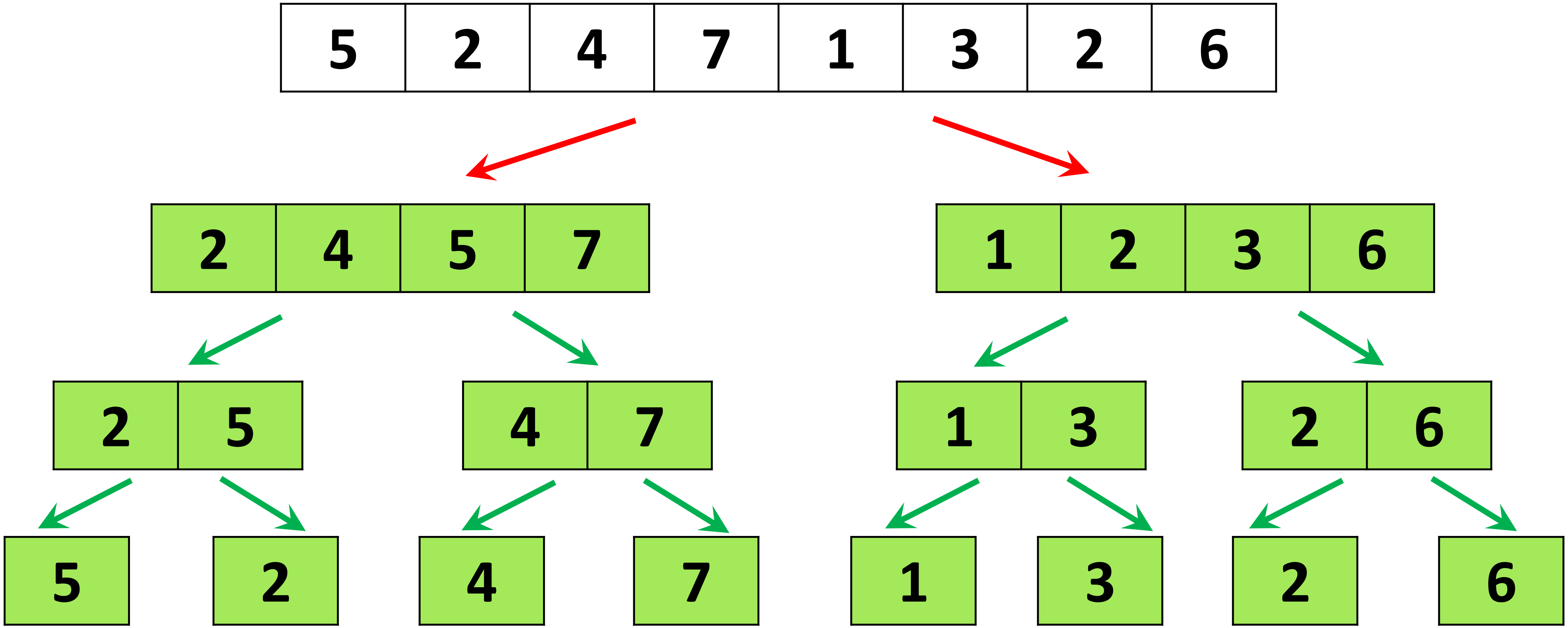
Merge-sort algorithm



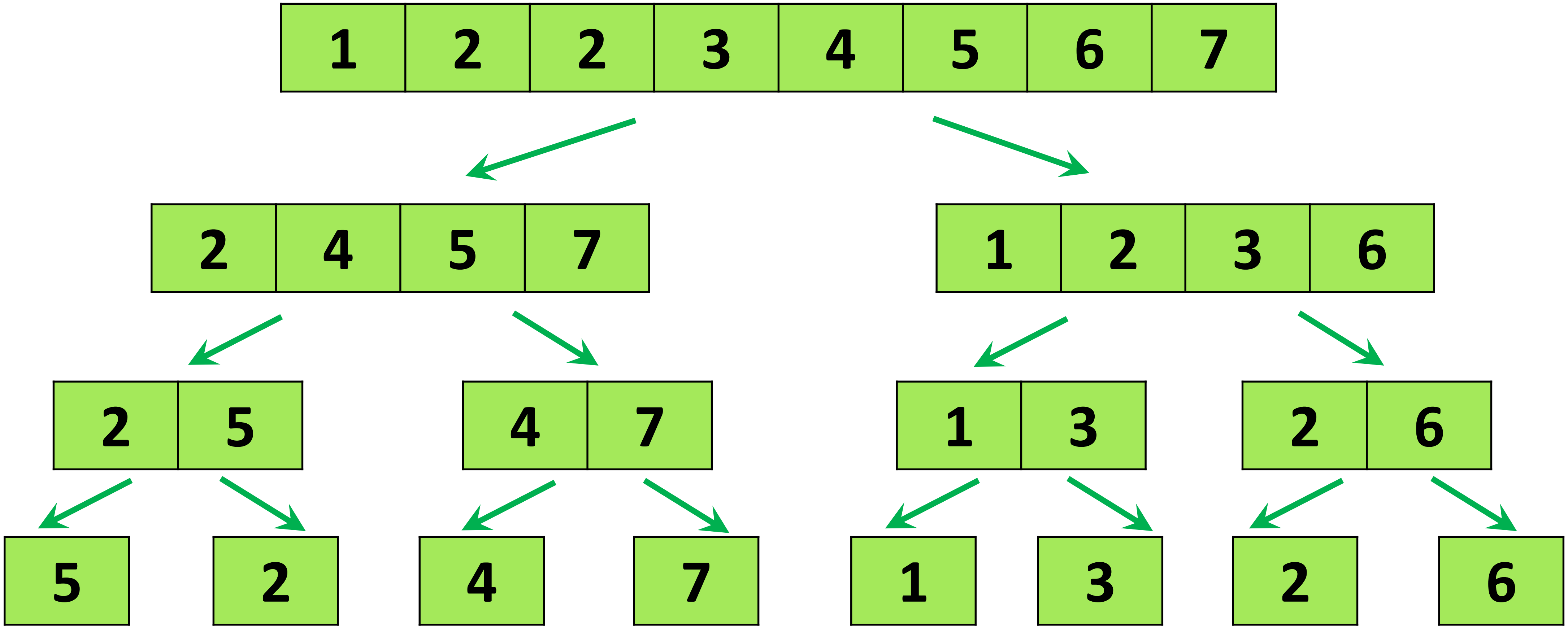
Merge-sort algorithm



Merge-sort algorithm



Merge-sort algorithm



Sorting using divide-and-conquer

```
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)
```

Sorting using divide-and-conquer

```
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)
```

Divide

Conquer

Combine

We call MERGE-SORT(A, 1, $A.length$) to sort the array

Sorting using divide-and-conquer

What if $p \geq r$?

$\text{MERGE-SORT}(A, p, r)$

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

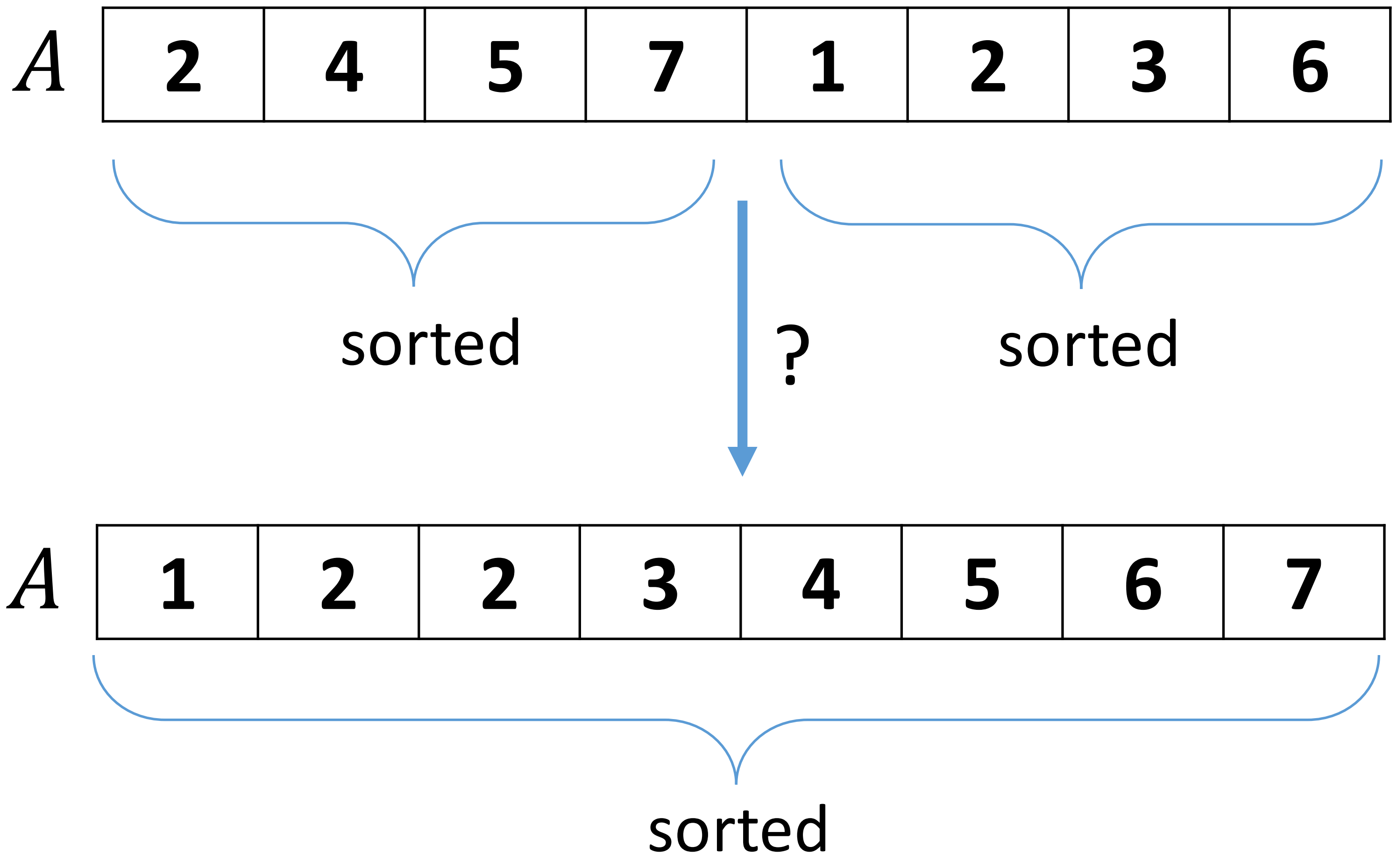
Divide

Conquer

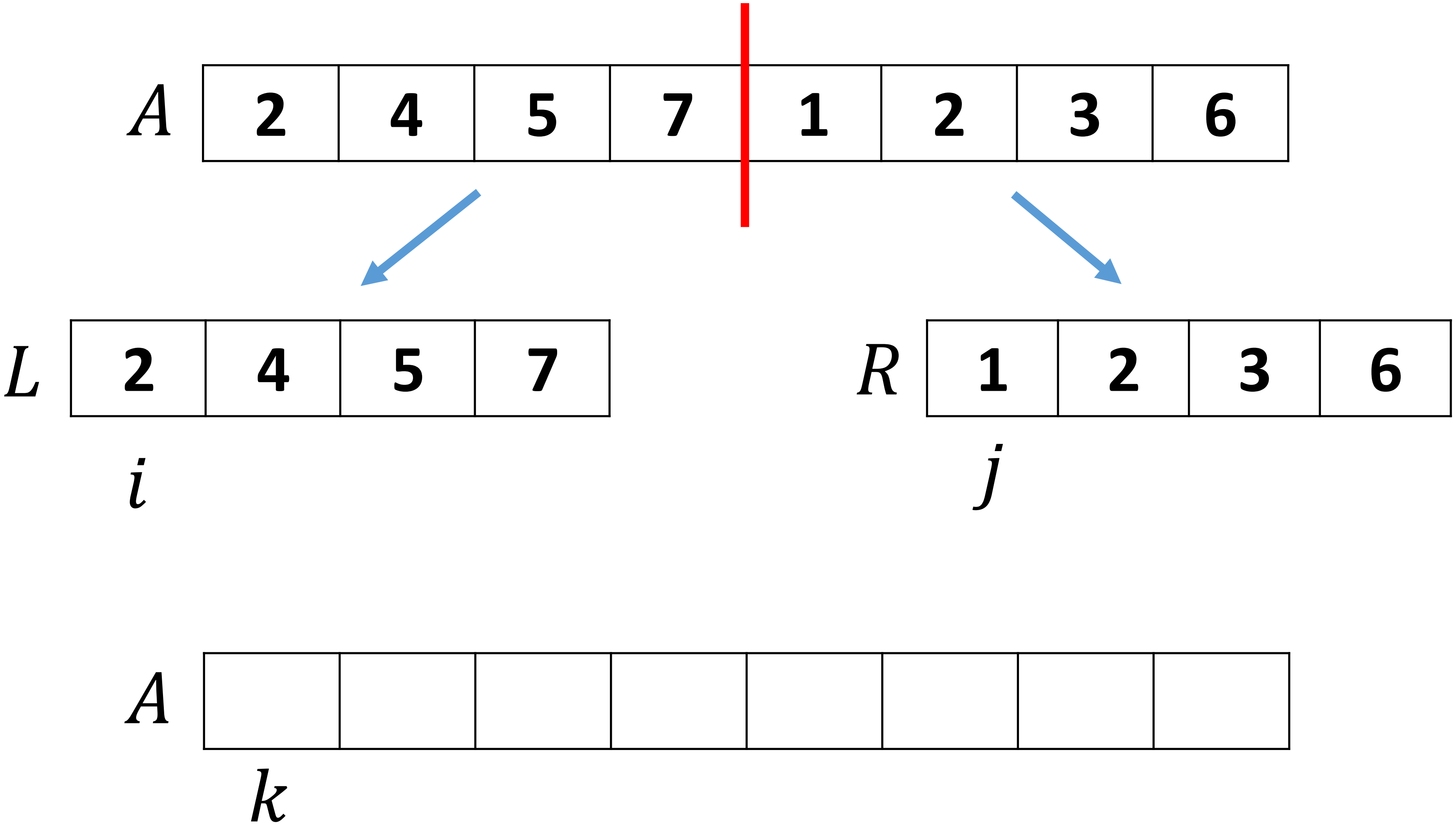
Combine

We call $\text{MERGE-SORT}(A, 1, A.length)$ to sort the array

How to merge efficiently?



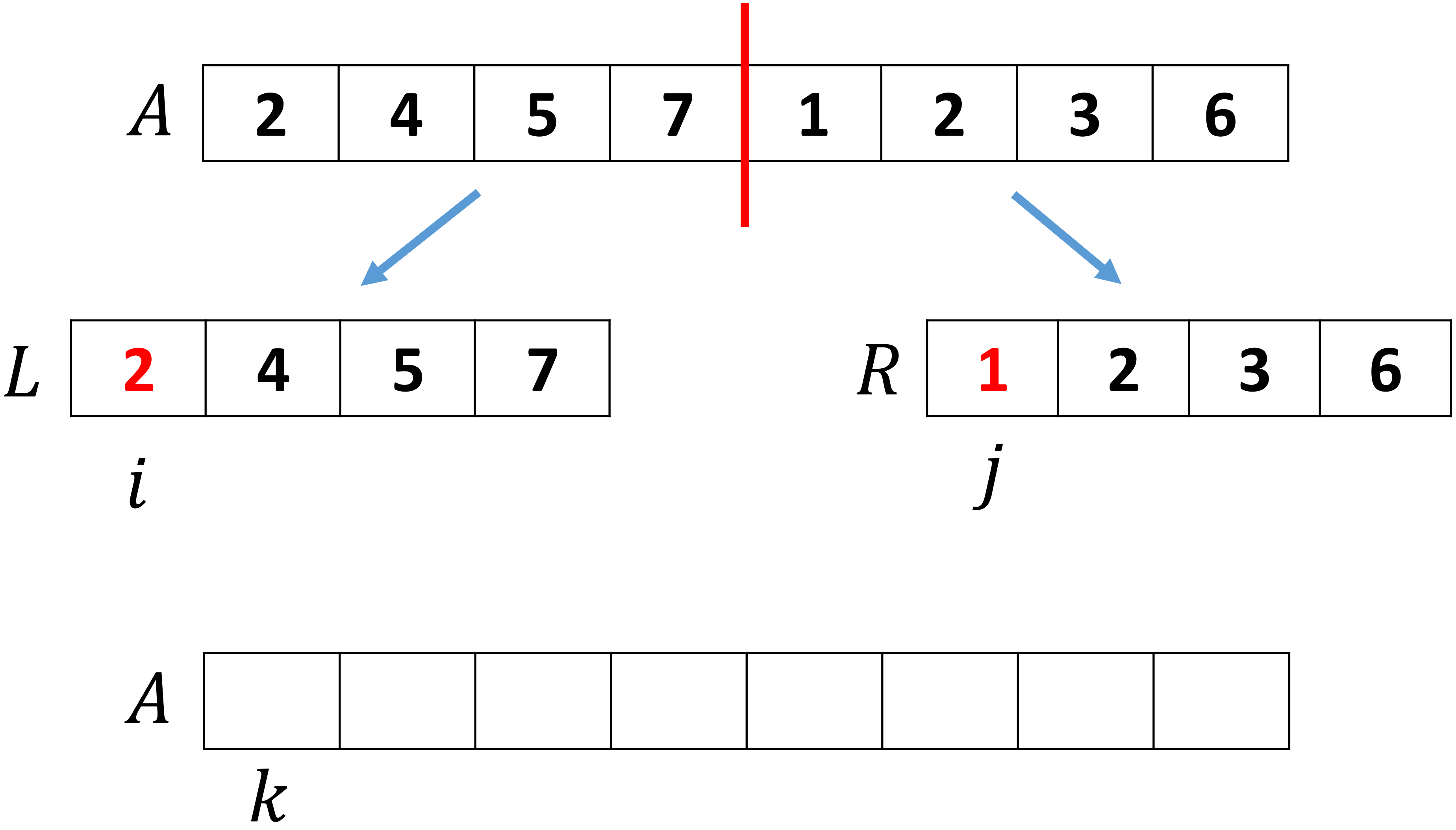
How to merge efficiently?



How to merge efficiently?

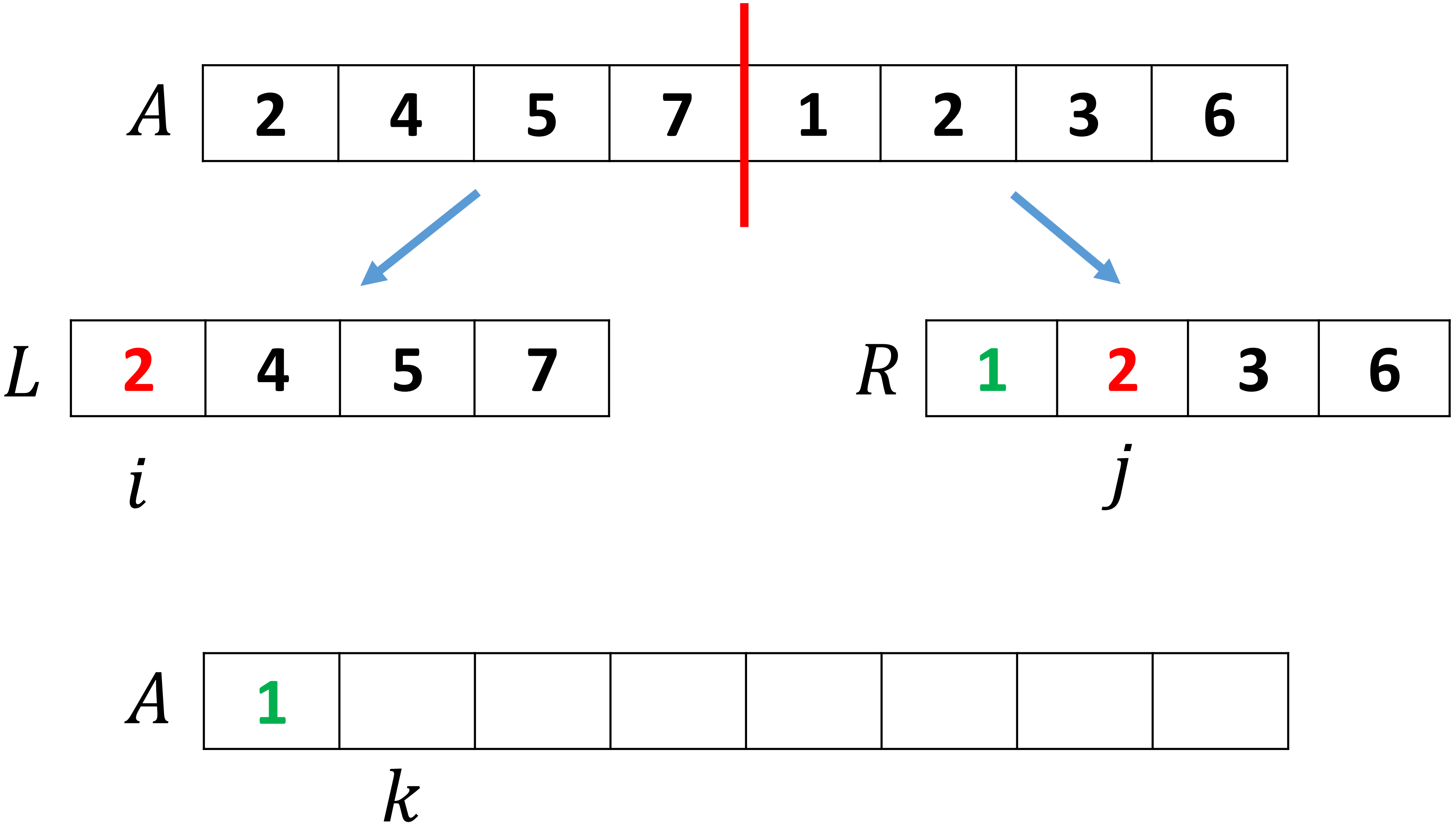
- The idea is as follows:
- Each time we compare $L[i]$ with $R[j]$, and put the smallest of these to at $A[k]$.
- Then, we increment the counter that had smallest value (either i or j) and we also increment k.
- Intuitively, this works because each time the minimum available number is put at $A[k]$. So, A, will eventually have all elements of L and R in the sorted order.

How to merge efficiently?

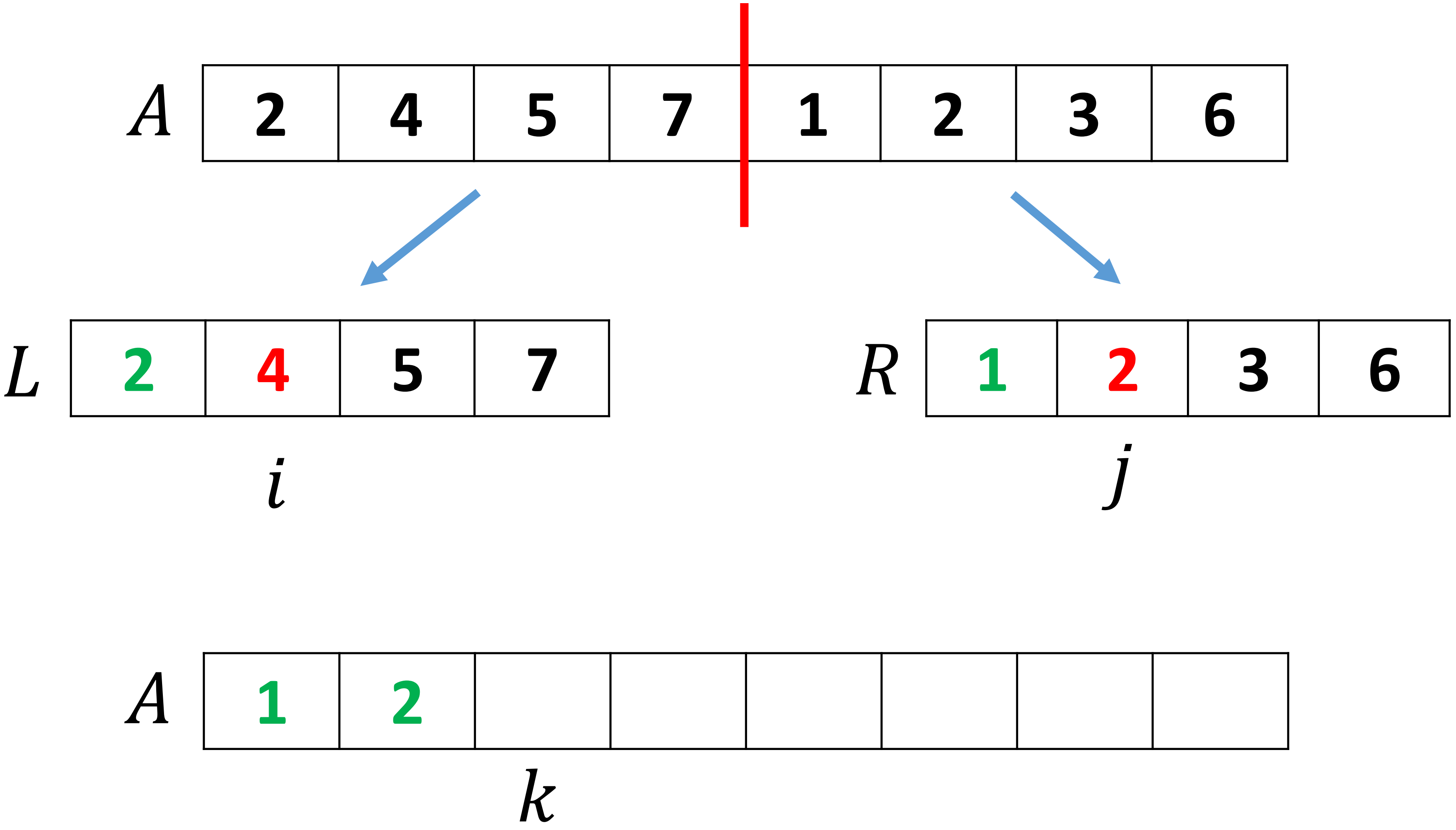


How to merge efficiently?

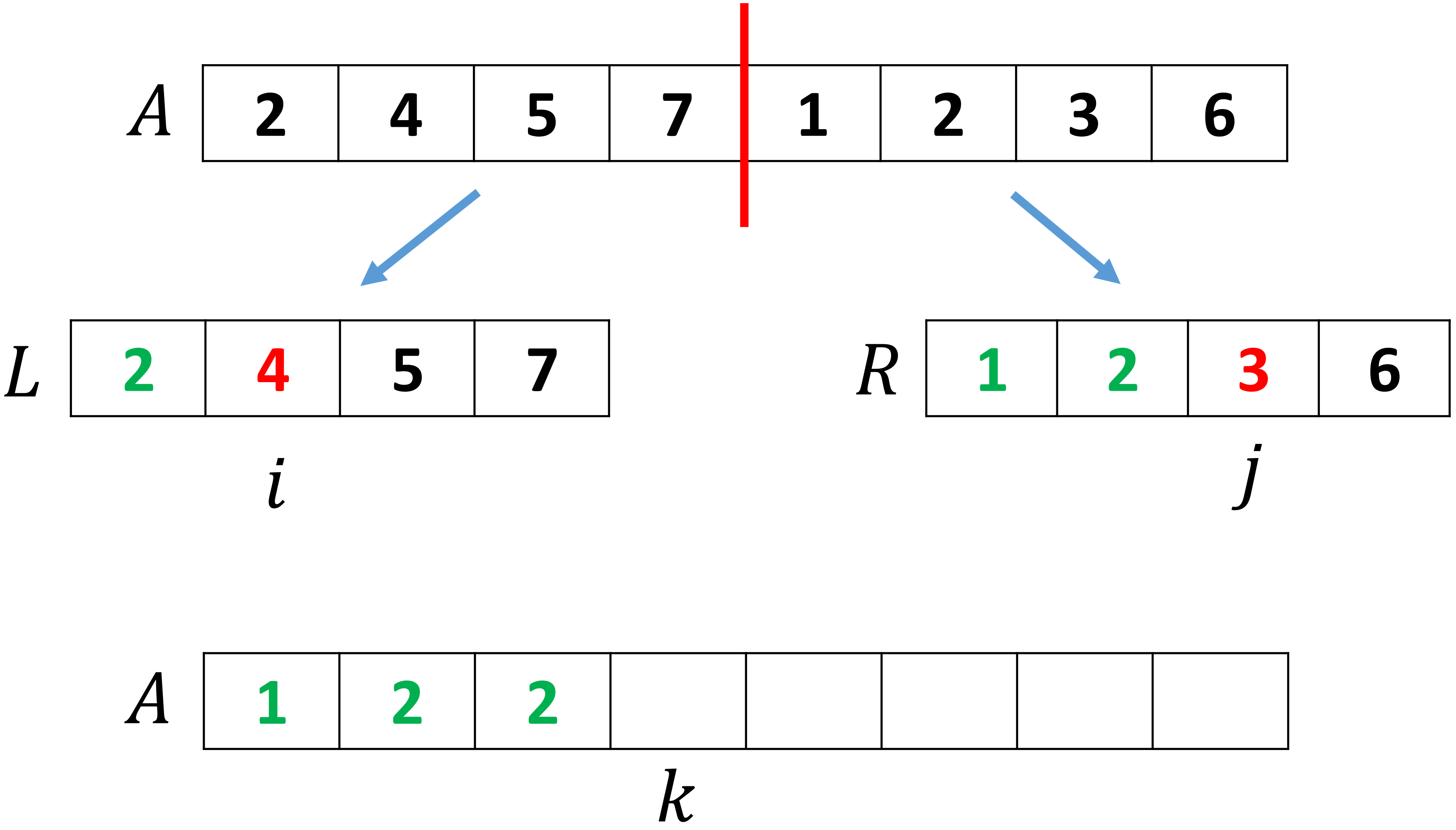
If $L[i] == R[j]$ we pick $L[i]$



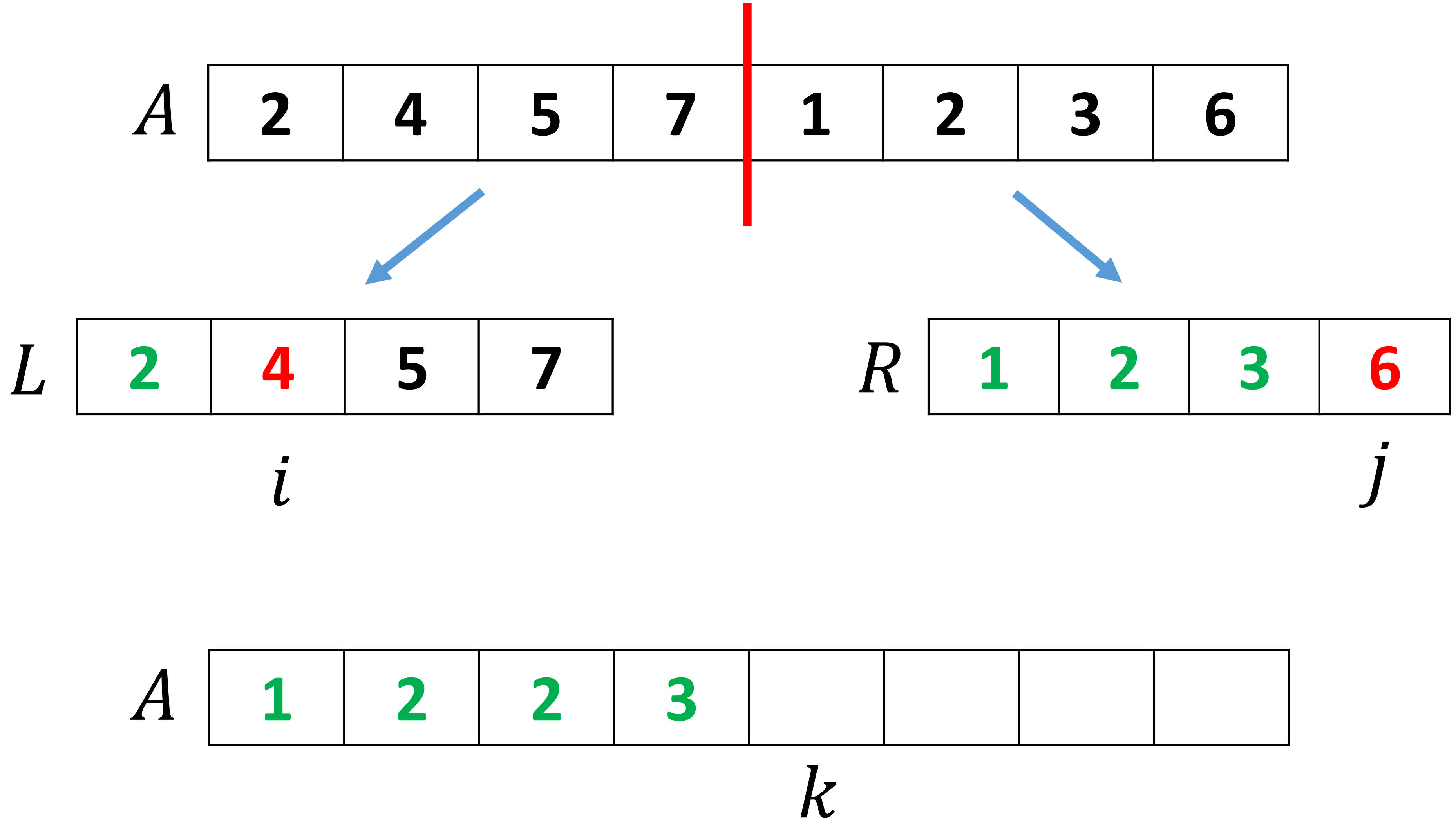
How to merge efficiently?



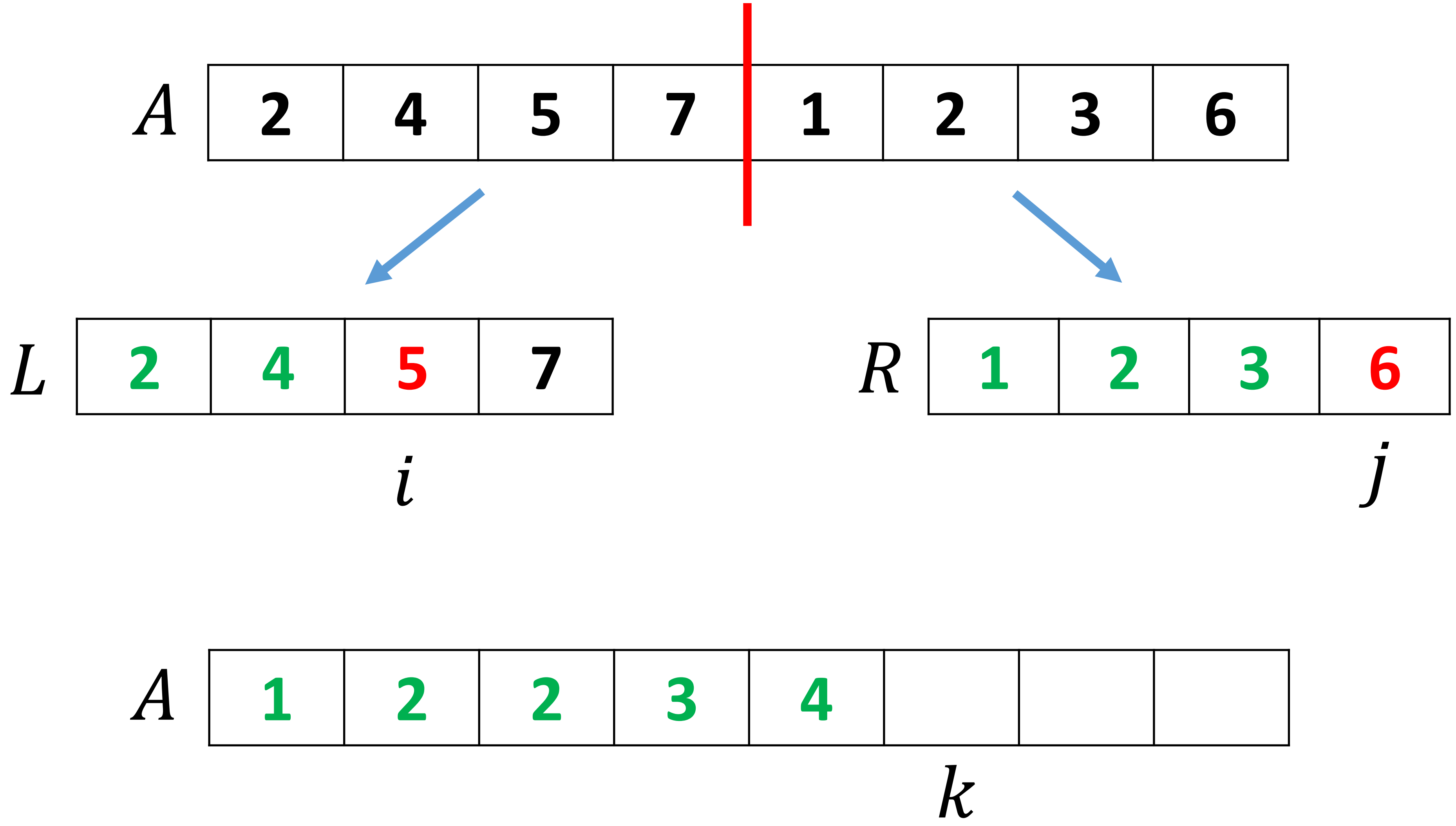
How to merge efficiently?



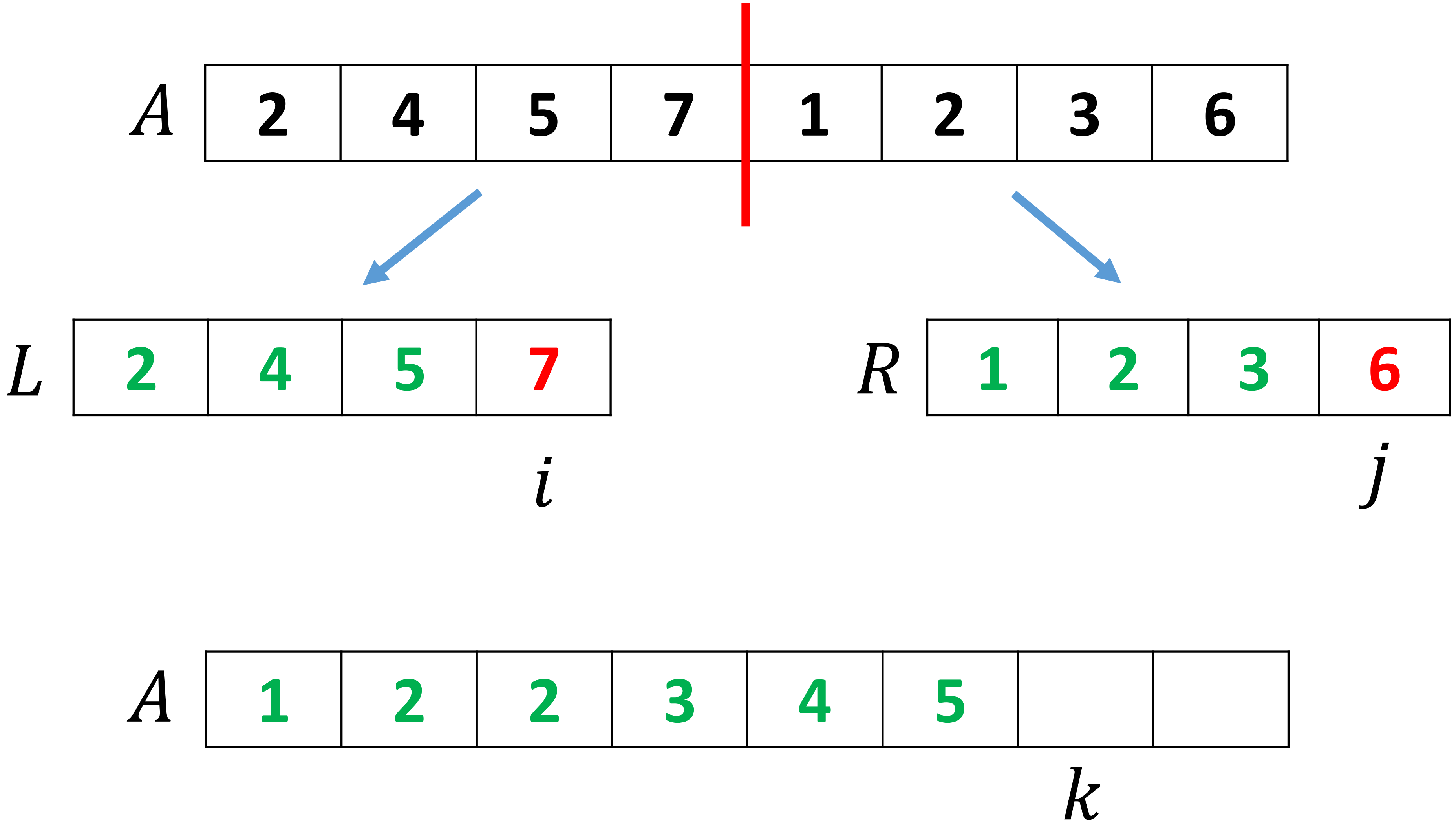
How to merge efficiently?



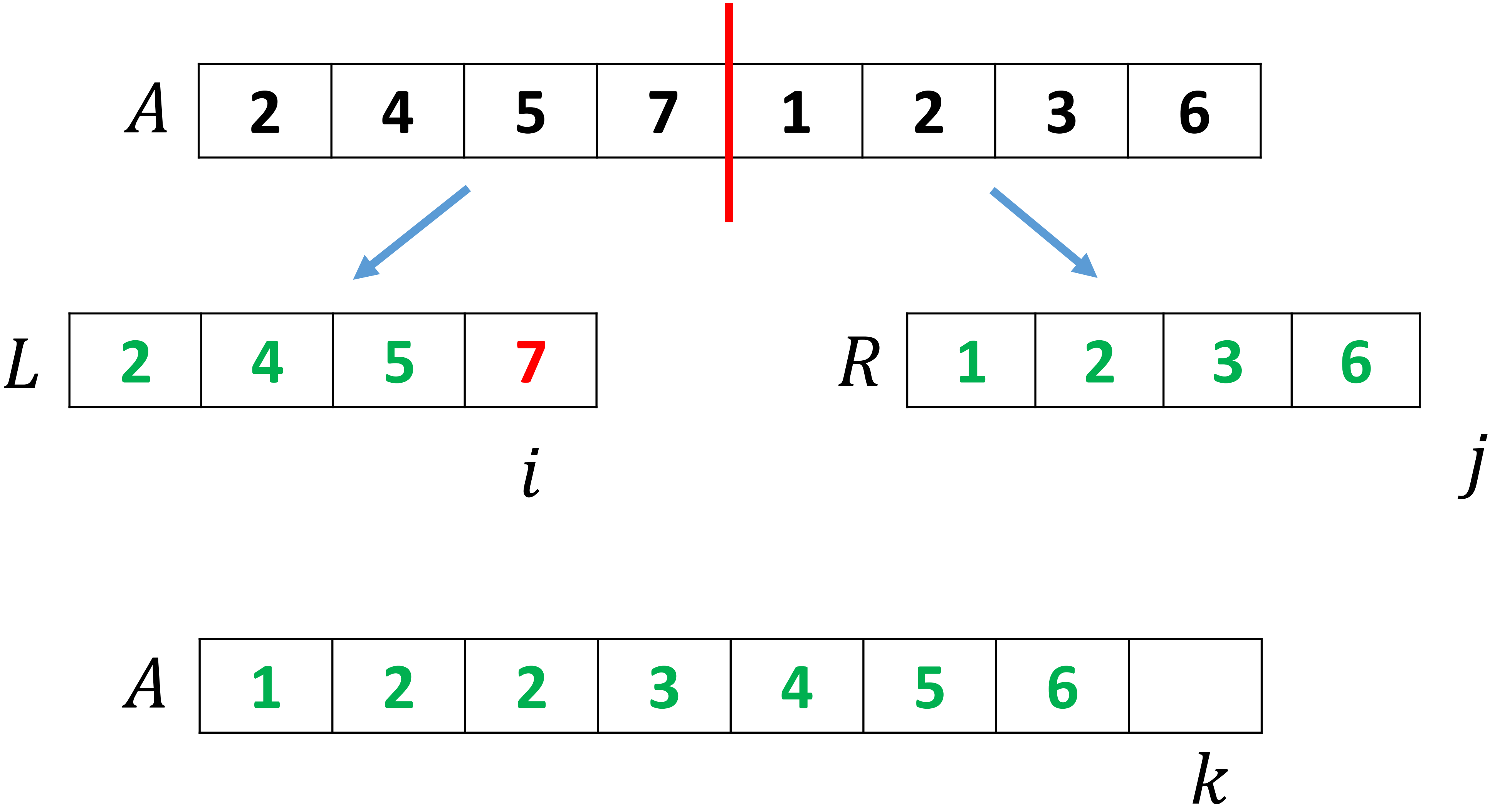
How to merge efficiently?



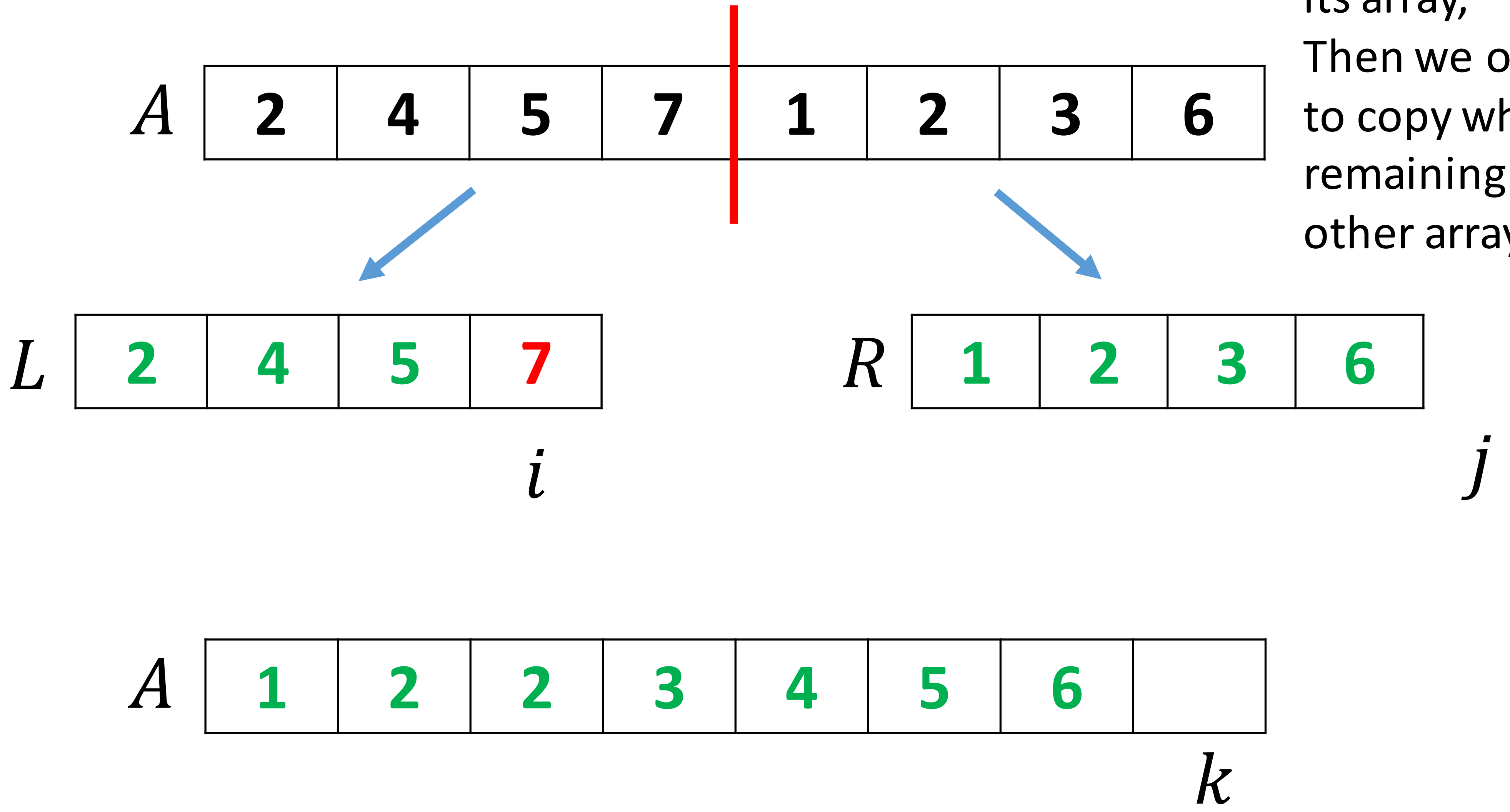
How to merge efficiently?



How to merge efficiently?

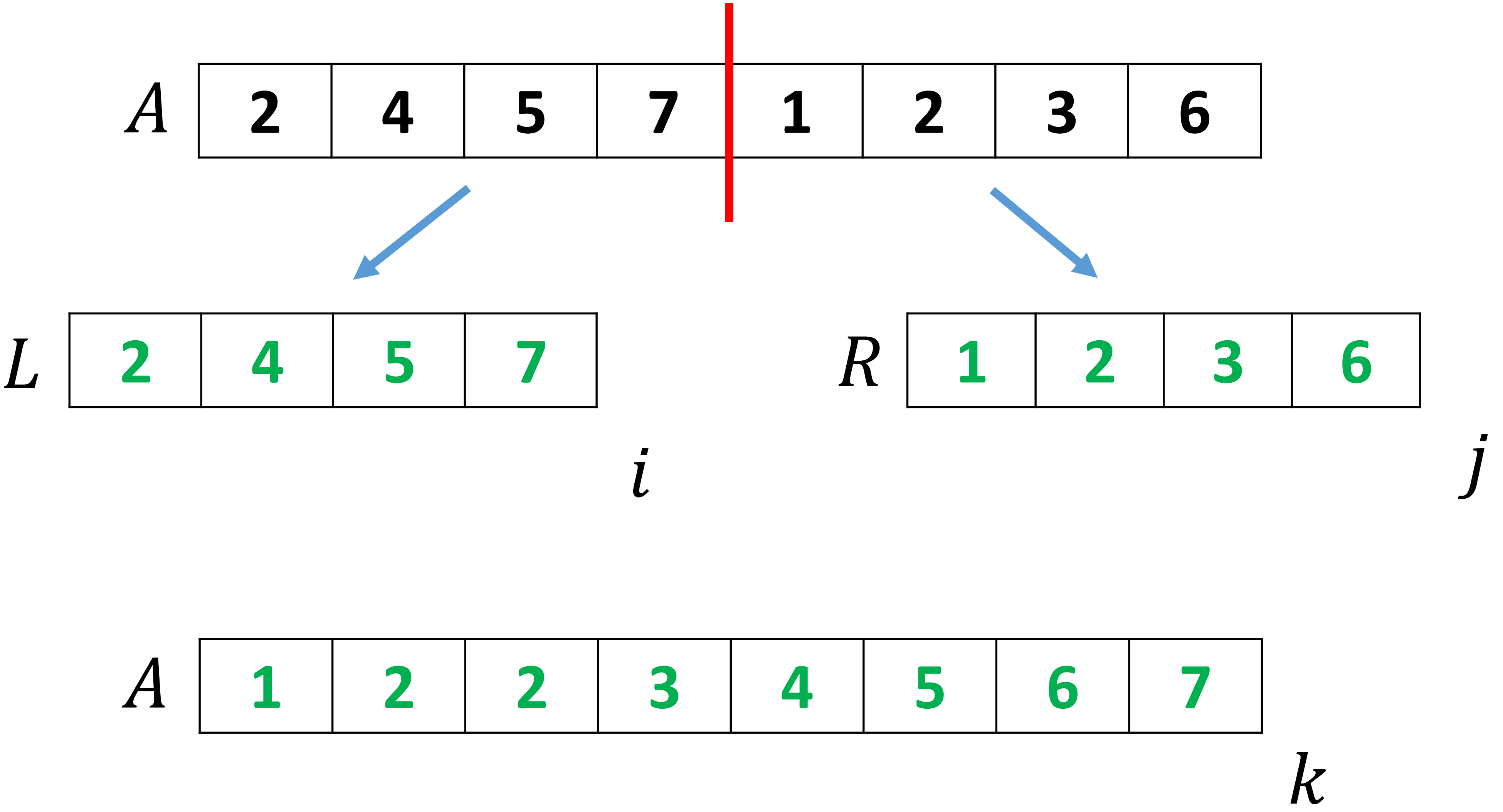


How to merge efficiently?



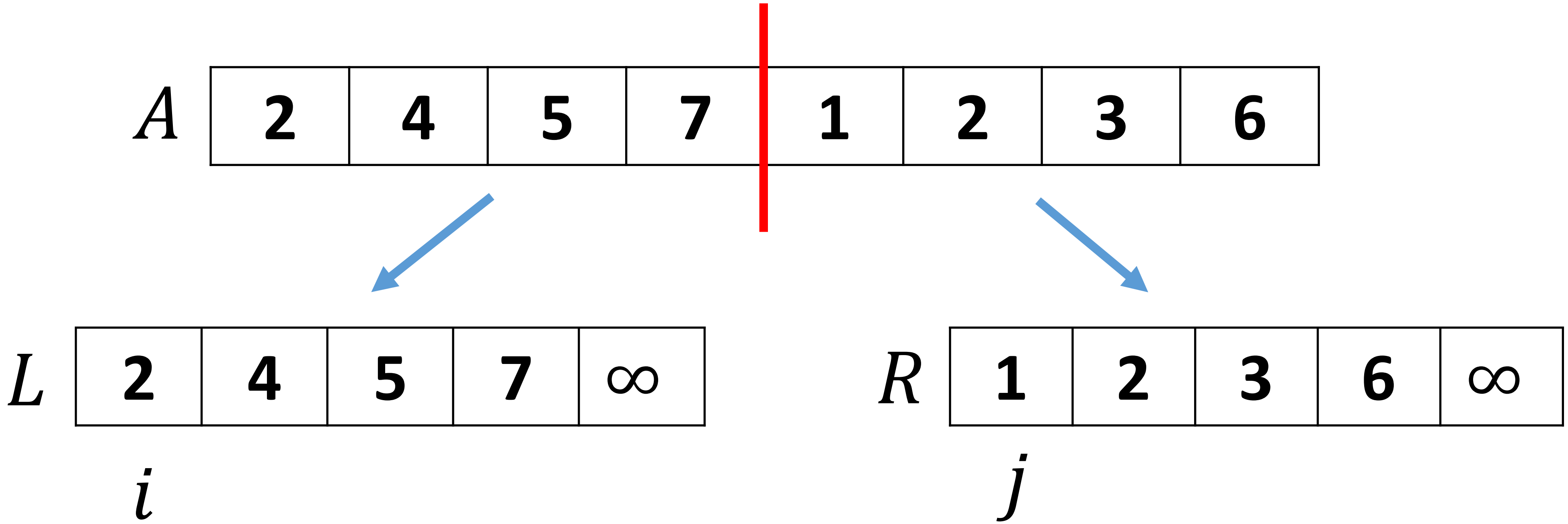
When one counter reaches the end of its array,
Then we only have to copy what is remaining from the other array

How to merge efficiently?



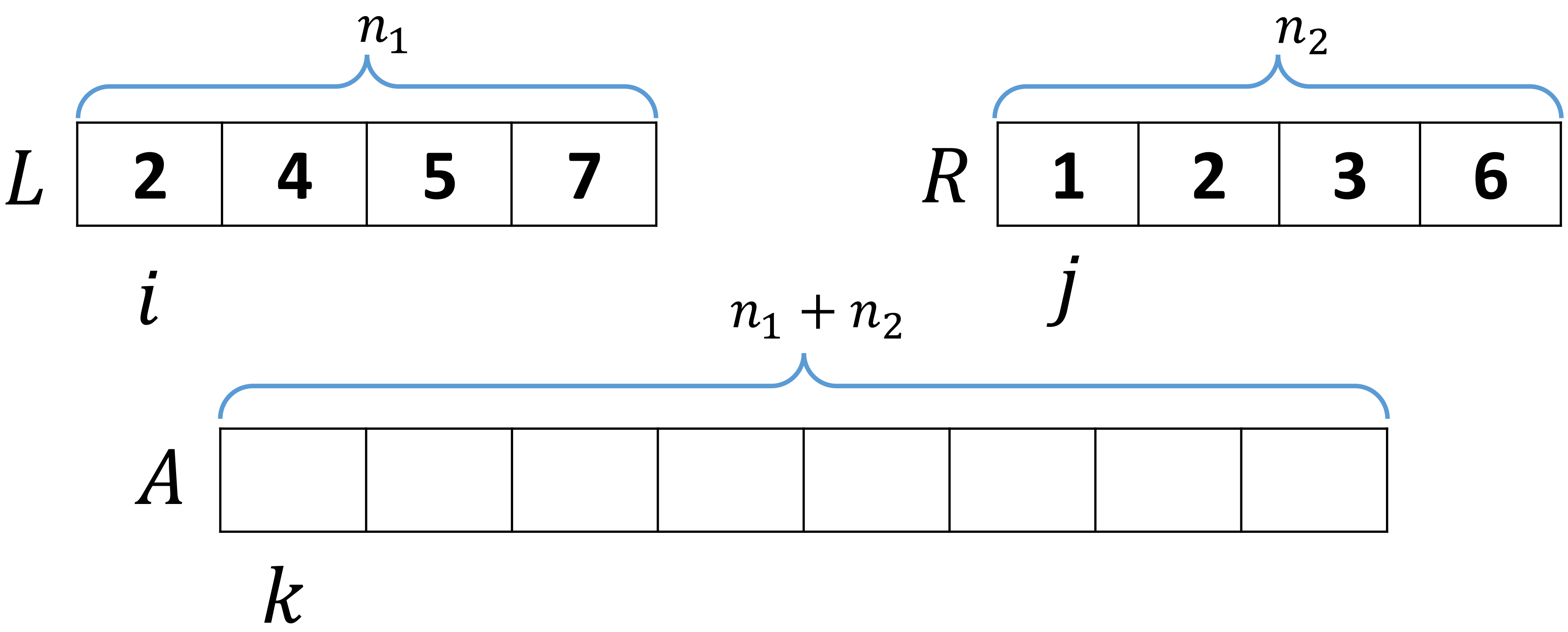
Implementation

- Pseudocode is in CLRS page 31 and uses <u>∞ **as sentinels** </u>to avoid checking whether i and j exceed the array's length.
- Sentinel means a **soldier who keeps watch**.
- In Java, we can use **Integer.MAX_VALUE** instead of ∞.



Analysis of merge

- **Question:** If L has size n_1 and R has size n_2, what is the complexity of merging them into an array of size $n_1 + n_2$?



Analysis of merge

- **Question:** If L has size n_1 and R has size n_2, what is the complexity of merging them into an array of size $n_1 + n_2$?
- **Answer:** $O(n_1 + n_2)$ because after $O(1)$ operations we copy a value from either R or L into A. As a result, k increments and we only do this until k reaches the end of A which has size $n_1 + n_2$.

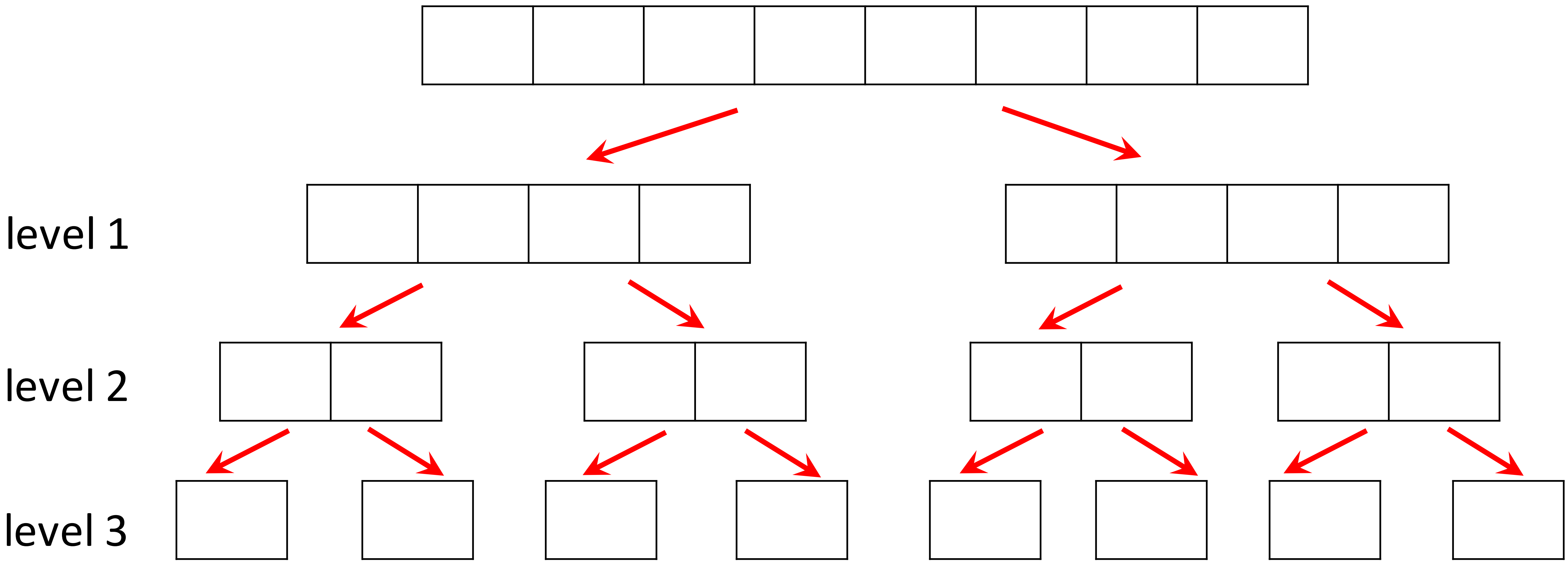
Analysis of merge

- **Question:** If L has size n_1 and R has size n_2, what is the complexity of merging them into an array of size $n_1 + n_2$?
- **Answer:** $O(n_1 + n_2)$ because after $O(1)$ operations we copy a value from either R or L into A. As a result, k increments and we only do this until k reaches the end of A which has size $n_1 + n_2$.

- **Note:** the running time is actually $\Theta(n_1 + n_2)$ but since we usually care about the worst-case analysis and finding an upper bound, we may use big-O.

Intuitive analysis of Merge-Sort

Intuitive analysis of MERGE-SORT



Intuitive analysis of Merge-Sort

- Since we are diving an array of size n by 2 until the size of the subarrays becomes 1, we need $\log n$ divisions.
- So, we will have $\mathrm{O}(\log n)$ levels.
- **At each level**, the time complexity for the merge algorithm **over all subarrays at that level,** is $\mathrm{O}(n)$.
- As a result, we need $\mathrm{O}(n \log n)$ time for the merge-sort.
- Note that we used the fact that $O(f)\,O(g) = O(f\,g)$.

Formal analysis of MERGE-SORT

Let's say $T(n)$ is the time needed for MERGE-SORT to sort an array of size n

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ 2\,T\left(\frac{n}{2}\right) + \Theta(n) & if\ n > 1 \end{cases}$$

```
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)
```

$T(n/2) \rightarrow$ MERGE-SORT(A, p, q)

$T(n/2) \rightarrow$ MERGE-SORT$(A, q+1, r)$

$\Theta(n) \rightarrow$ MERGE(A, p, q, r)