Algorithms & Data Structures

Lecture 03
Divide-and-Conquer: Merge Sort

Giovanni Bacci
giovbacci@cs.aau.dk

Outline

- Algorithm Design: Divide and Conquer

- Merge sort (Correctness & Runtime Analysis)

Intended Learning Goals

KNOWLEDGE

- Mathematical reasoning on concepts such as recursion, induction, concrete and abstract computational complexity
- Data structures, algorithm principles e.g., search trees, hash tables, dynamic programming, divide-and-conquer
- Graphs and graph algorithms e.g., graph exploration, shortest path, strongly connected components.

SKILLS

- Determine abstract complexity for specific algorithms
- Perform complexity and correctness analysis for simple algorithms
- Select and apply appropriate algorithms for standard tasks

COMPETENCES

- Ability to face a non-standard programming assignment
- Develop algorithms and data structures for solving specific tasks
- Analyse developed algorithms

Divide and Conquer

Divide and rule (Latin: *divide et impera*), or **divide and conquer**, in politics and sociology is gaining and maintaining power by breaking up larger concentrations of power into pieces that individually have less power than the one implementing the strategy.[citation needed]

The maxim ***divide et impera*** has been attributed to Philip II of Macedon. It was utilised by the Roman ruler Julius Caesar and the French emperor Napoleon (together with the maxim *divide ut regnes*)

…form Wikipedia

It is also and effective algorithmic strategy!

Divide & Conquer

Basic idea

- **Divide** the problem into a number of disjoint* subproblems that are smaller instances of the same problem
- **Conquer** the subproblem by solving them recursively. If the subproblem is small (and easy) enough solve it trivially
- **Combine** the solution of the subproblems into the solution for the original problem

(*) we will consider the case where the subproblems overlap each other in Lecture 09

Example: Binary Search

The element search problem:
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ sorted in non-decreasing order, two indices l, r such that $1 \leq l \leq r \leq n$, and a number a to search
Output: An index $l \leq i \leq r$ such that $a = a_i$ if there exists such an index, 0 otherwise.

```
BIN-SEARCH(A, l, r, a)
 1  if l < r
 2        m = ⌊(l + r)/2⌋
 3        if A[m] ≥ a
 4              return BIN-SEARCH(A, l, m, a)
 5        else
 6              return BIN-SEARCH(A, m + 1, r, a)
 7  elseif l = r and A[l] = a
 8        return l
 9  else
10        return 0
```

Example: Binary Search

The element search problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ sorted in non-decreasing order, two indices l, r such that $1 \leq l \leq r \leq n$, and a number a to search

Output: An index $l \leq i \leq r$ such that $a = a_i$ if there exists such an index, 0 otherwise.

```
BIN-SEARCH(A, l, r, a)
 1  if l < r
 2        m = ⌊(l + r)/2⌋
 3        if A[m] ≥ a
 4              return BIN-SEARCH(A, l, m, a)
 5        else
 6              return BIN-SEARCH(A, m + 1, r, a)
 7  elseif l = r and A[l] = a
 8        return l
 9  else
10        return 0
```

Trivial case: the subarray has only one element

Example: Binary Search

The element search problem:
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ sorted in non-decreasing order, two indices l, r such that $1 \leq l \leq r \leq n$, and a number a to search
Output: An index $l \leq i \leq r$ such that $a = a_i$ if there exists such an index, 0 otherwise.

```
BIN-SEARCH(A, l, r, a)
 1  if l < r
 2        m = ⌊(l + r)/2⌋                          Divide
 3        if A[m] ≥ a
 4              return BIN-SEARCH(A, l, m, a)
 5        else
 6              return BIN-SEARCH(A, m + 1, r, a)
 7  elseif l = r and A[l] = a
 8        return l                                 Trivial case:
 9  else                                           the subarray has
10        return 0                                 only one element
```

Example: Binary Search

The element search problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ sorted in non-decreasing order, two indices l, r such that $1 \leq l \leq r \leq n$, and a number a to search

Output: An index $l \leq i \leq r$ such that $a = a_i$ if there exists such an index, 0 otherwise.

```
BIN-SEARCH(A, l, r, a)
 1  if l < r
 2        m = ⌊(l + r)/2⌋
 3        if A[m] ≥ a
 4              return BIN-SEARCH(A, l, m, a)
 5        else
 6              return BIN-SEARCH(A, m + 1, r, a)
 7  elseif l = r and A[l] = a
 8        return l
 9  else
10        return 0
```

Divide

Conquer

Trivial case: the subarray has only one element

Example: Binary Search

The element search problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ sorted in non-decreasing order, two indices l, r such that $1 \leq l \leq r \leq n$, and a number a to search

Output: An index $l \leq i \leq r$ such that $a = a_i$ if there exists such an index, 0 otherwise.

```
BIN-SEARCH(A, l, r, a)
 1  if l < r
 2        m = ⌊(l + r)/2⌋
 3        if A[m] ≥ a
 4              return BIN-SEARCH(A, l, m, a)
 5        else
 6              return BIN-SEARCH(A, m + 1, r, a)
 7  elseif l = r and A[l] = a
 8        return l
 9  else
10        return 0
```

Divide

Conquer

Combine

Trivial case: the subarray has only one element

Sorting Problem

The sorting problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ of the input such that $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

- We have seen *Insertion Sort*
- It uses an **incremental approach**: having sorted the subarray $A[1..j-1]$, we inserted the single element $A[j]$ into its proper place, yielding the sorted subarray $A[1..j]$.
- Worst-case running time: $\Theta(n^2)$

Merge Sort

- Efficient sorting algorithm (we'll see this later)
- Works by **dividing the problem into smaller and more manageable sub-problems**

The main idea

- **Divide:** split the n-element sequence to be sorted into two subsequences of $n/2$ elements each (*)
- **Conquer:** sort the two subsequences recursively using merge sort.
- **Combine:** merge the two sorted subsequences to produce the sorted answer.

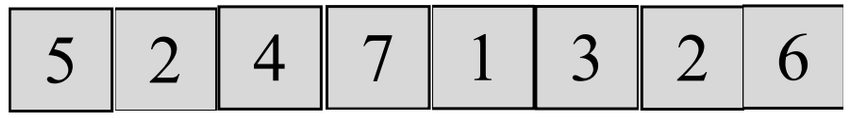
(*) The recursion "bottoms out" when the sequence has length 1, since every sequence of length 1 is already in sorted order.

Merge Sort

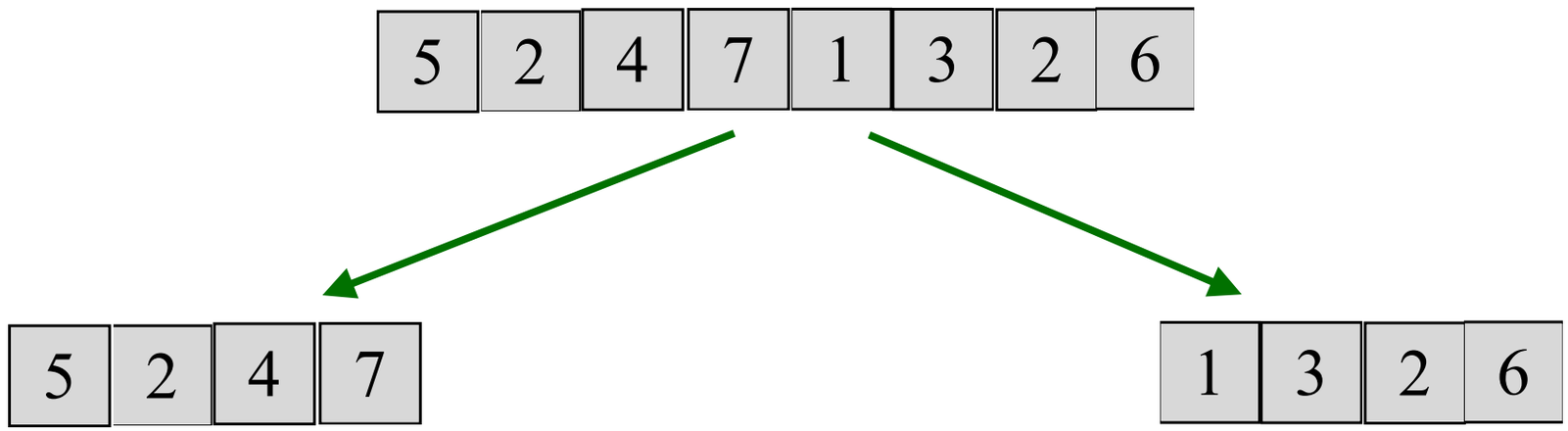
- The procedure takes an array $A[1..n]$ and two indices p and r such that $1 \leq p \leq r \leq n$.
- When the execution terminates the subarray $A[p\,..\,r]$ is sorted
- To sort entire array we simply call MERGE-SORT$(A,1,n)$

```
MERGE-SORT(A, p, r)
1   if p < r
2       q = ⌊(p + r)/2⌋                    Divide
3       MERGE-SORT(A, p, q)                Conquer
4       MERGE-SORT(A, q + 1, r)            Conquer
5       MERGE(A, p, q, r)                  Combine
```

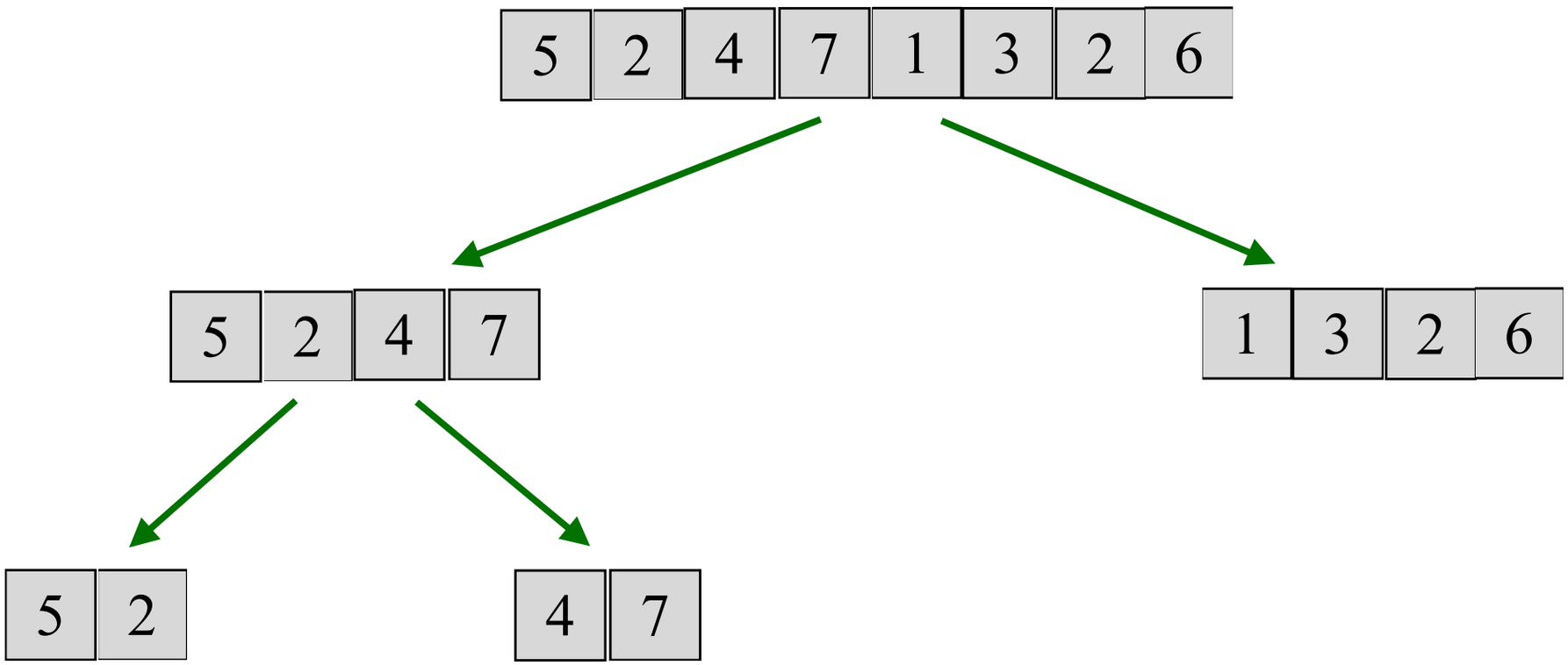
Merge Sort: Example



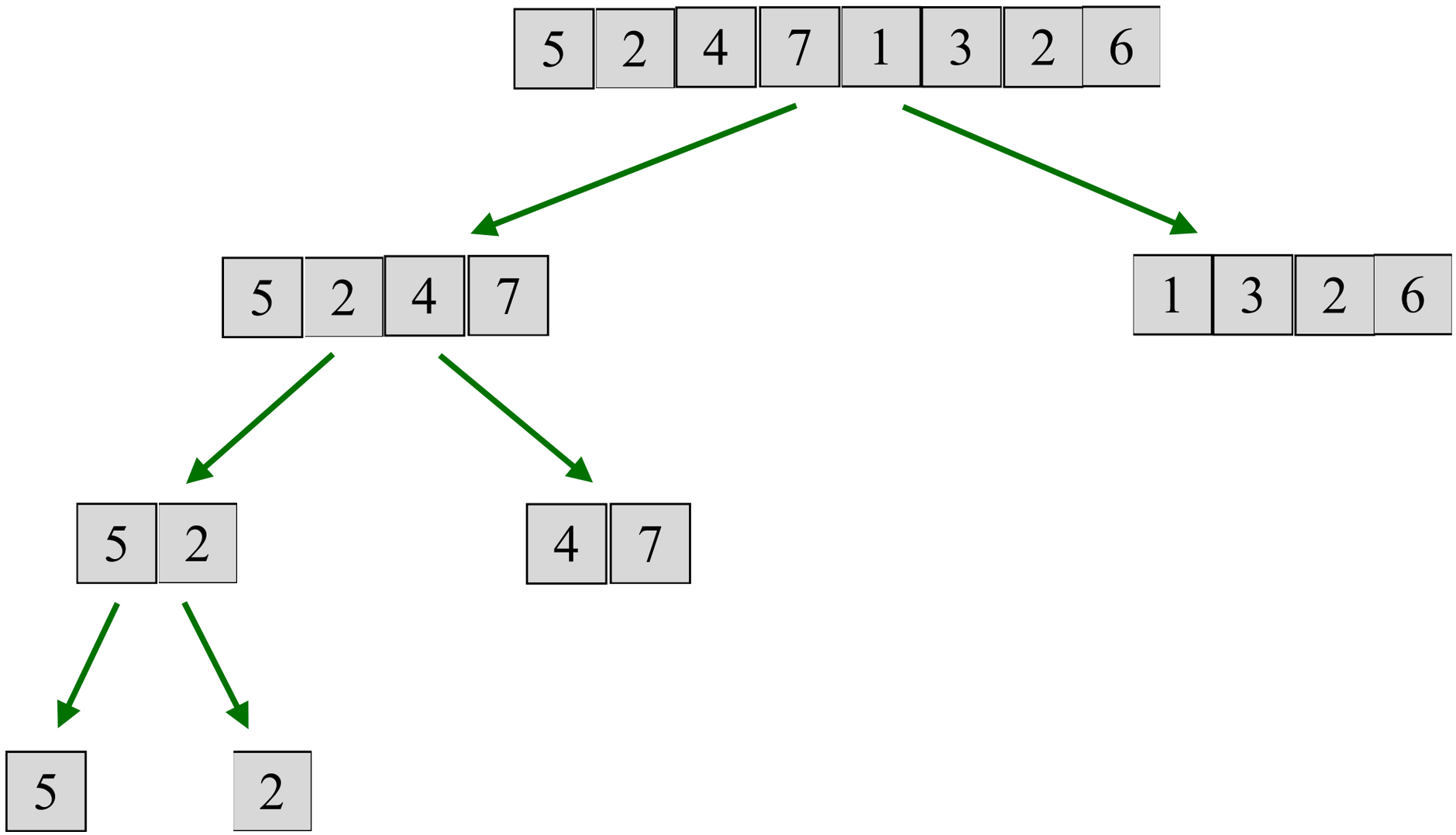
Merge Sort: Example



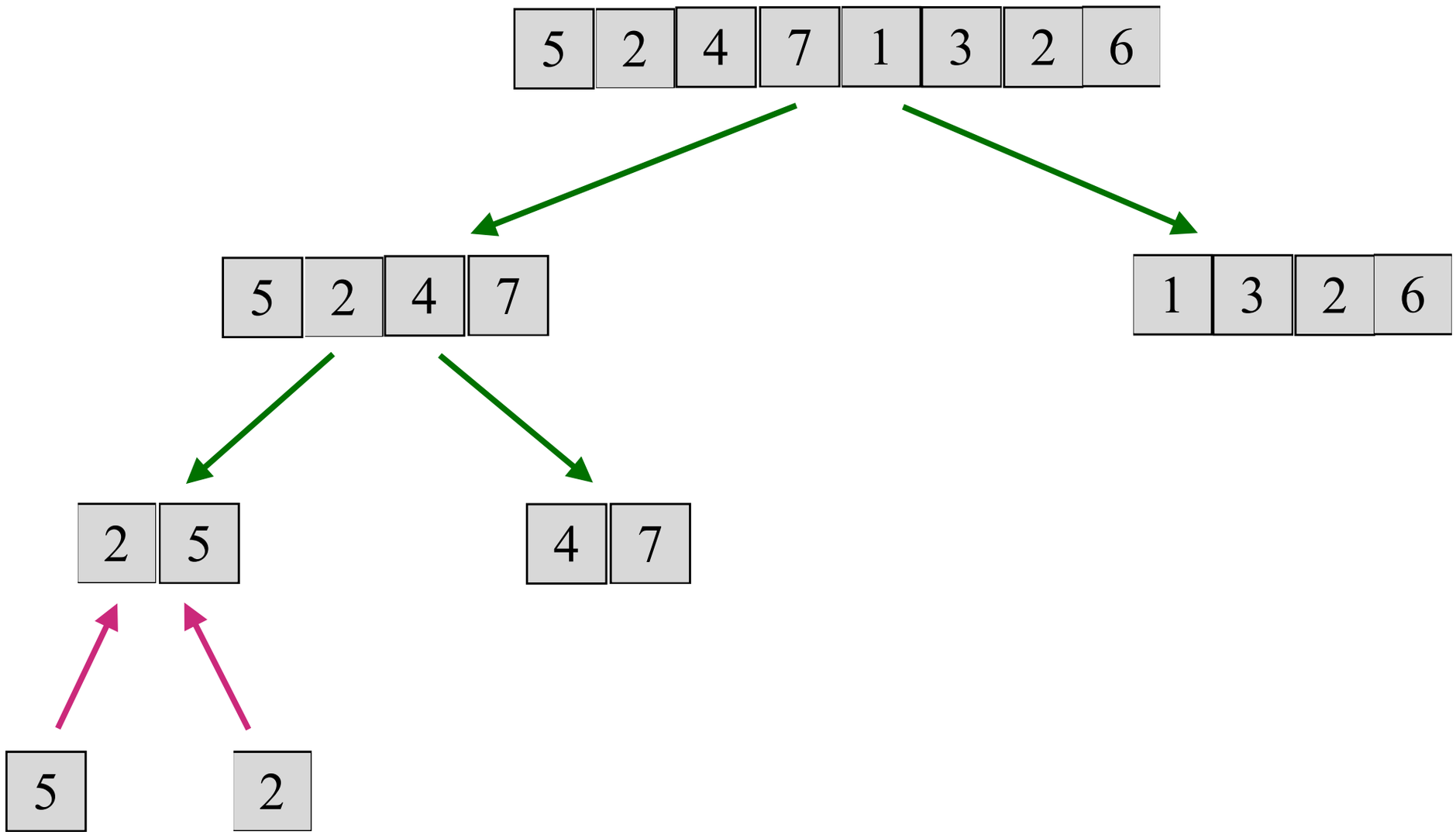
Merge Sort: Example



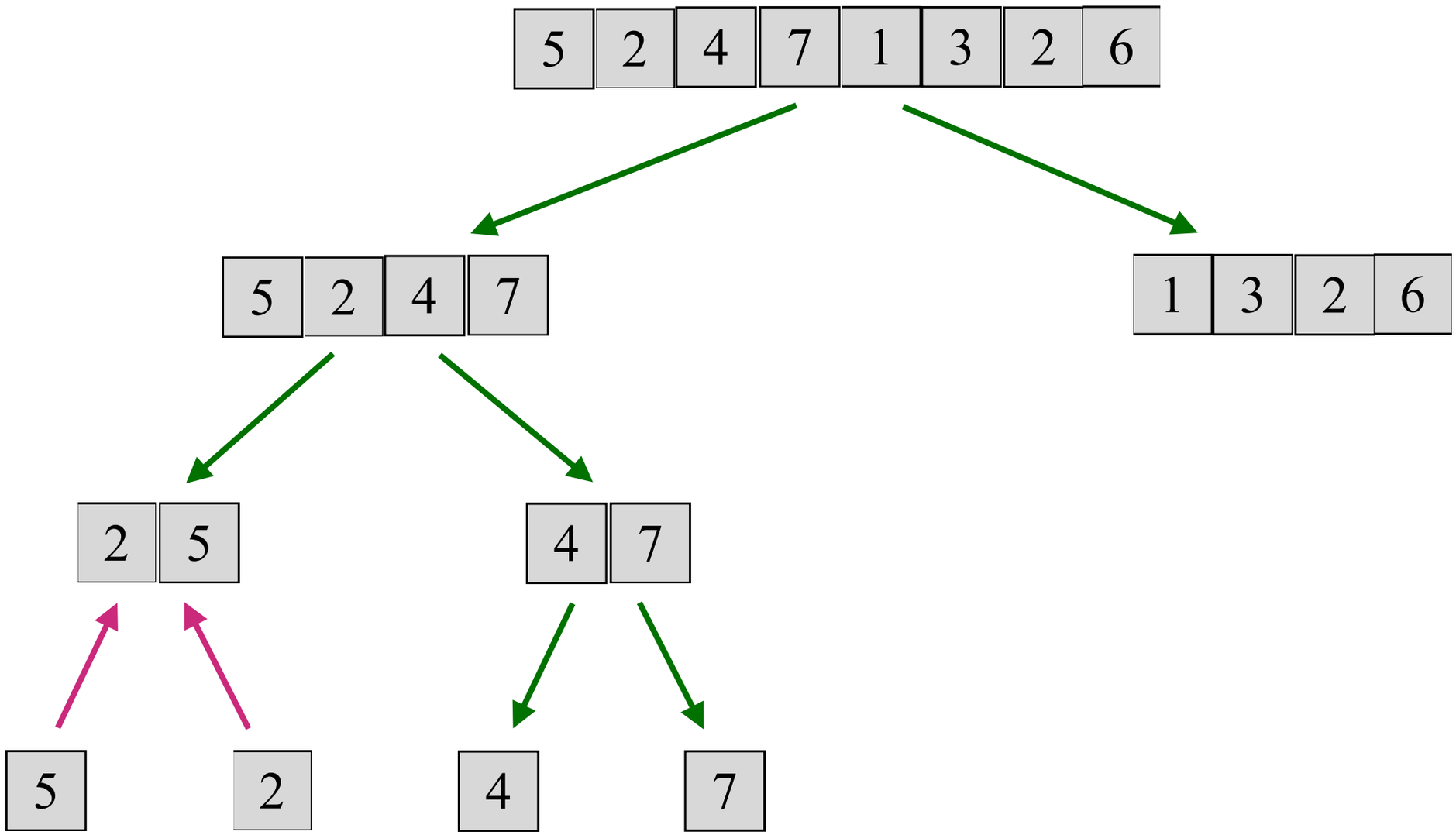
Merge Sort: Example



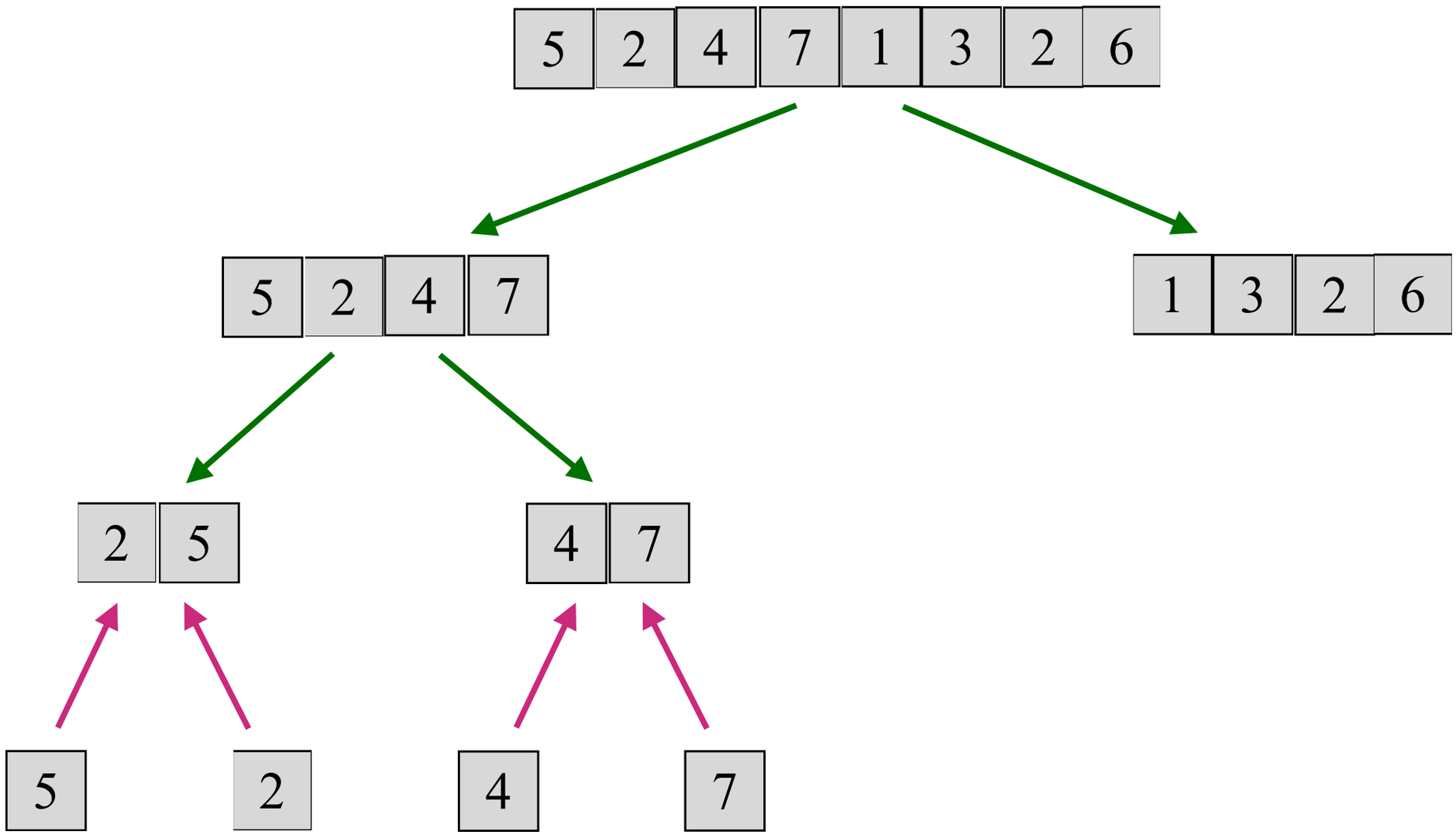
Merge Sort: Example



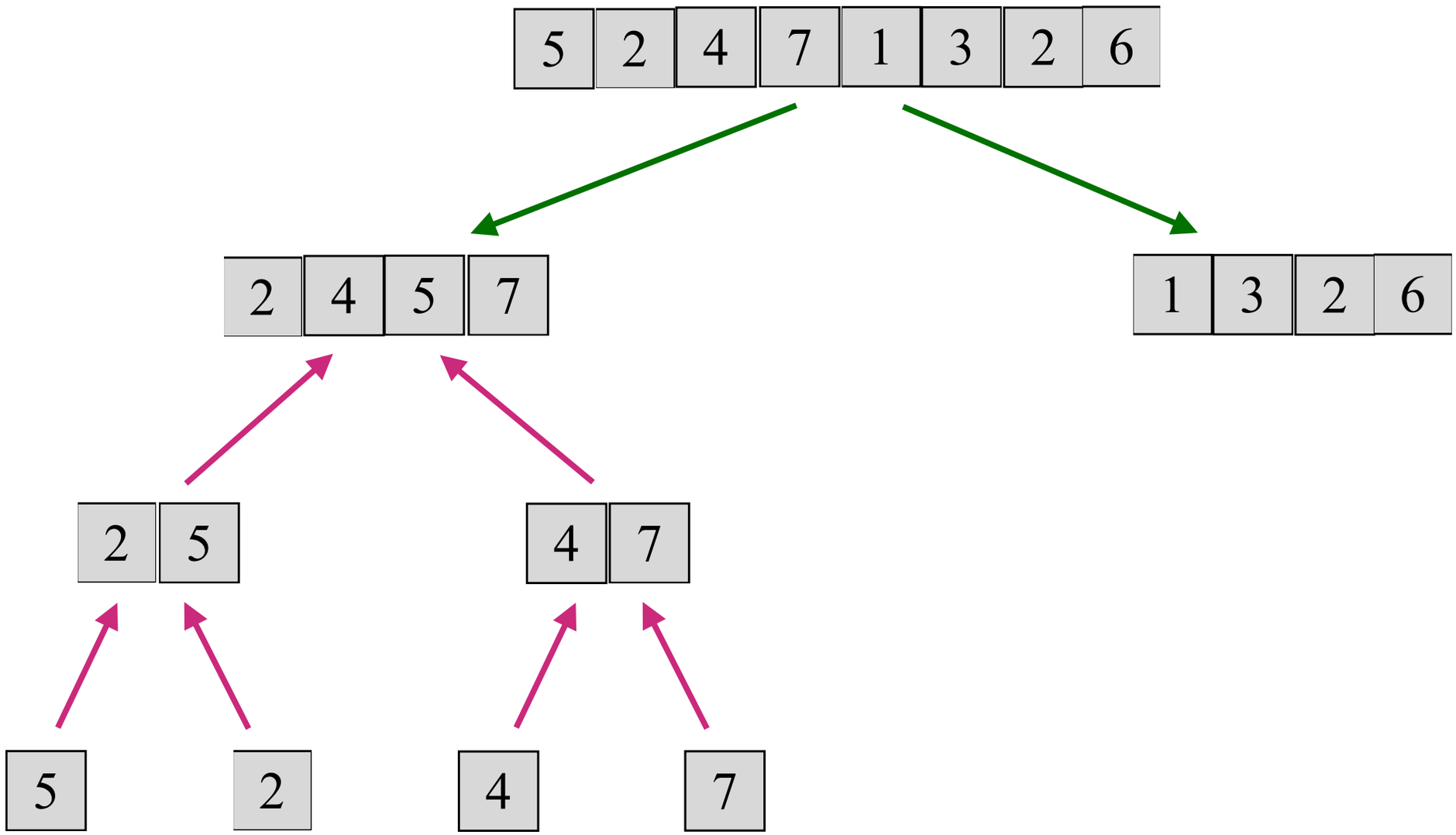
Merge Sort: Example



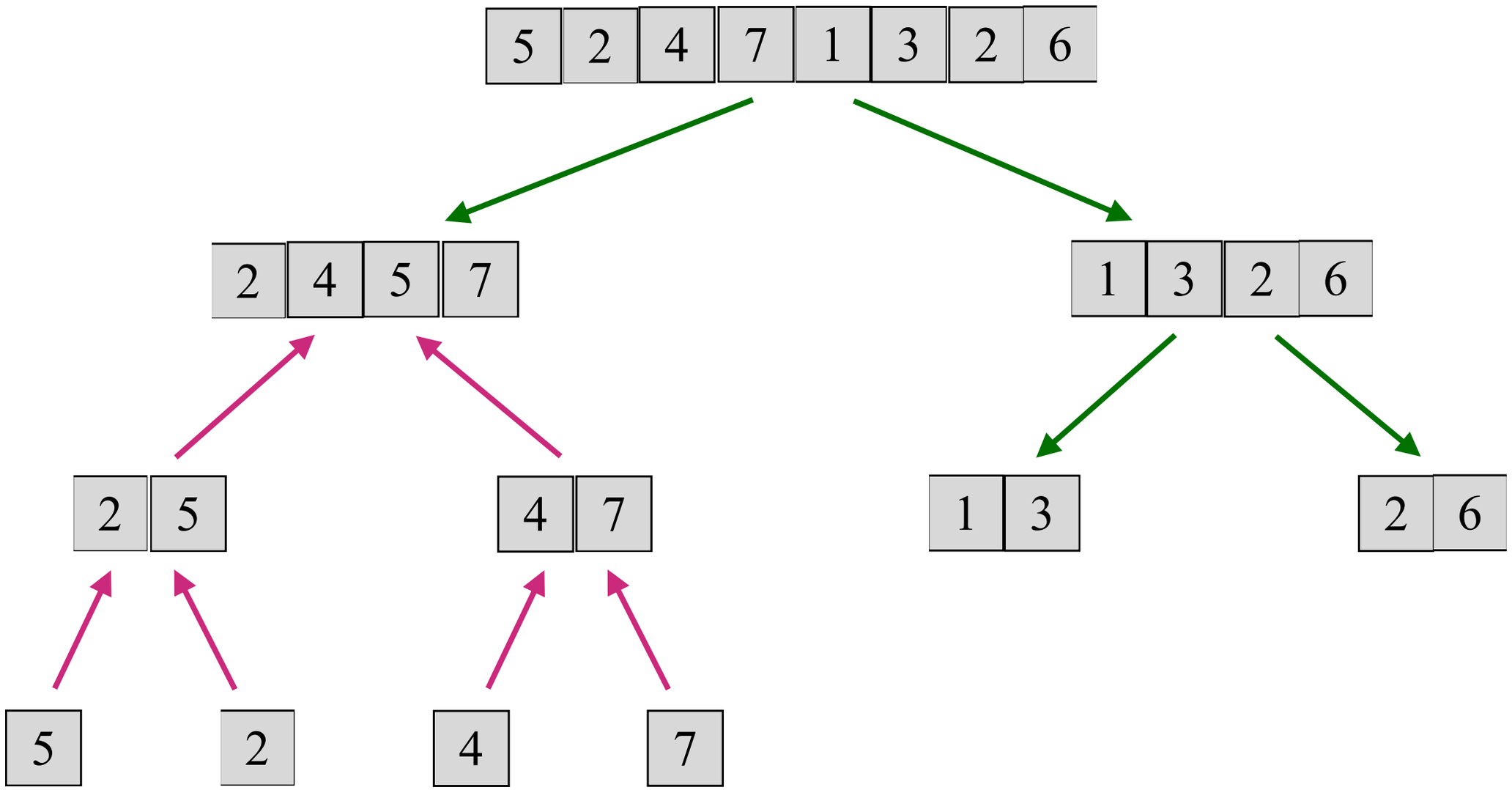
Merge Sort: Example



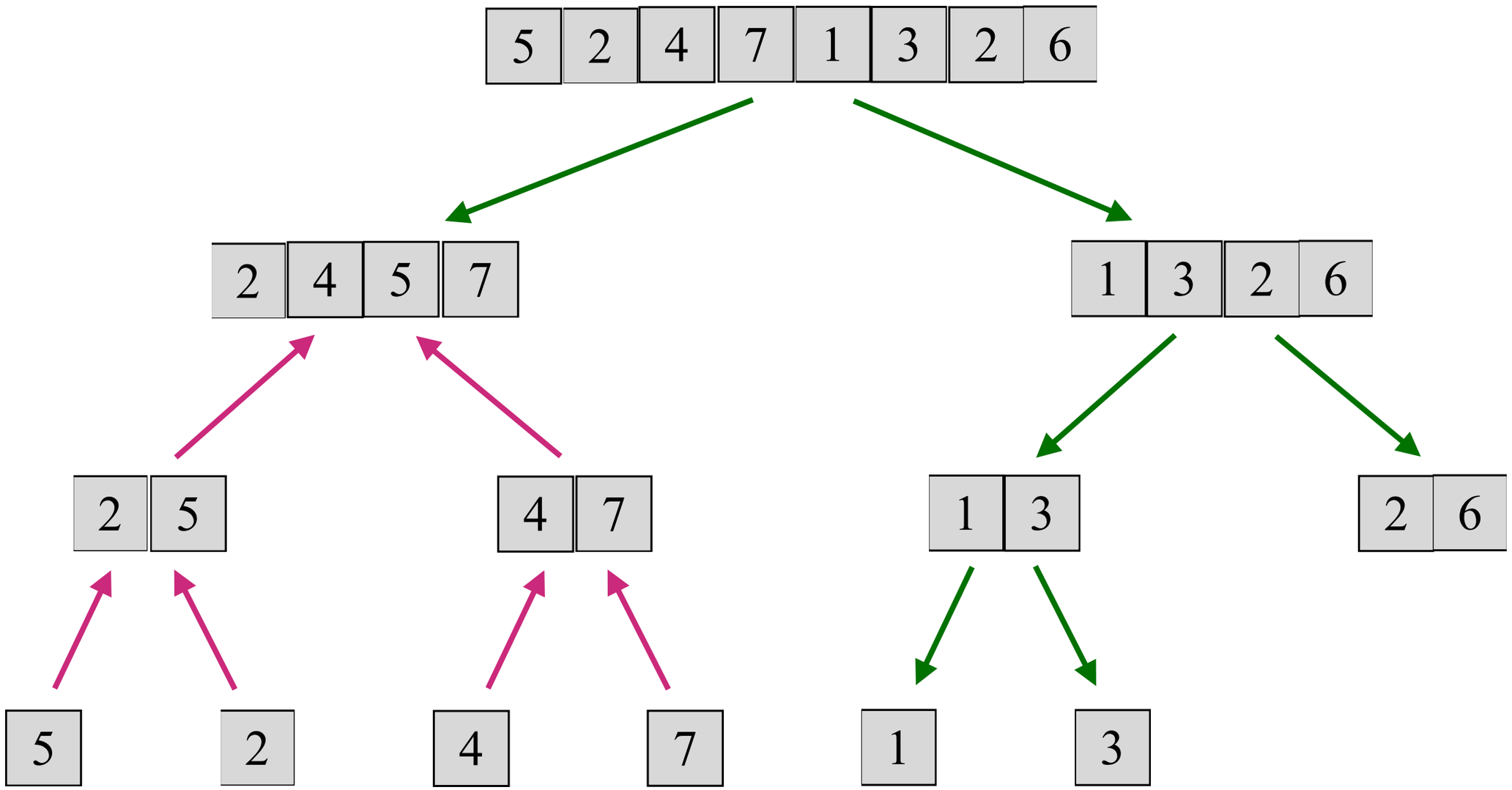
Merge Sort: Example



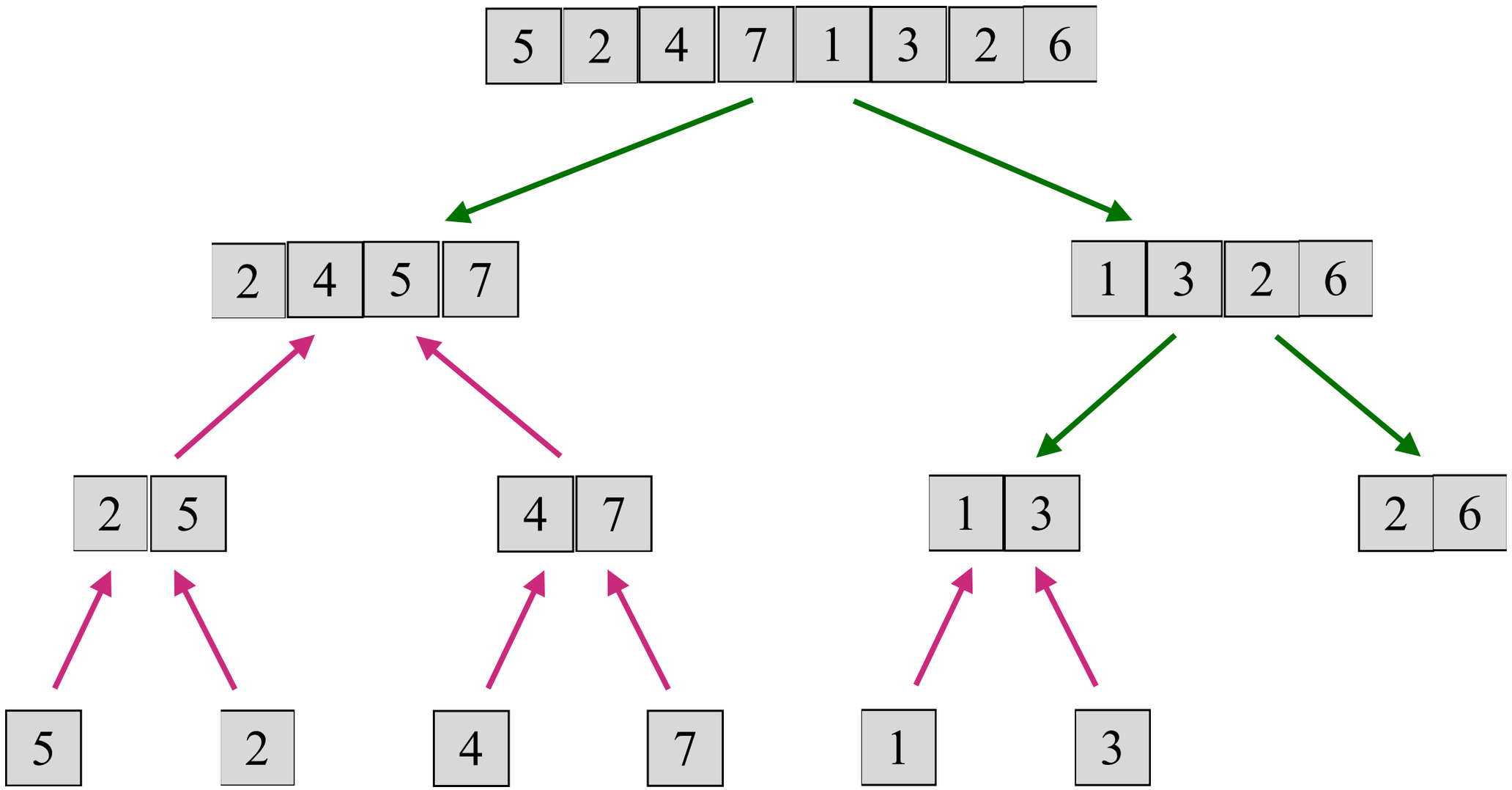
Merge Sort: Example



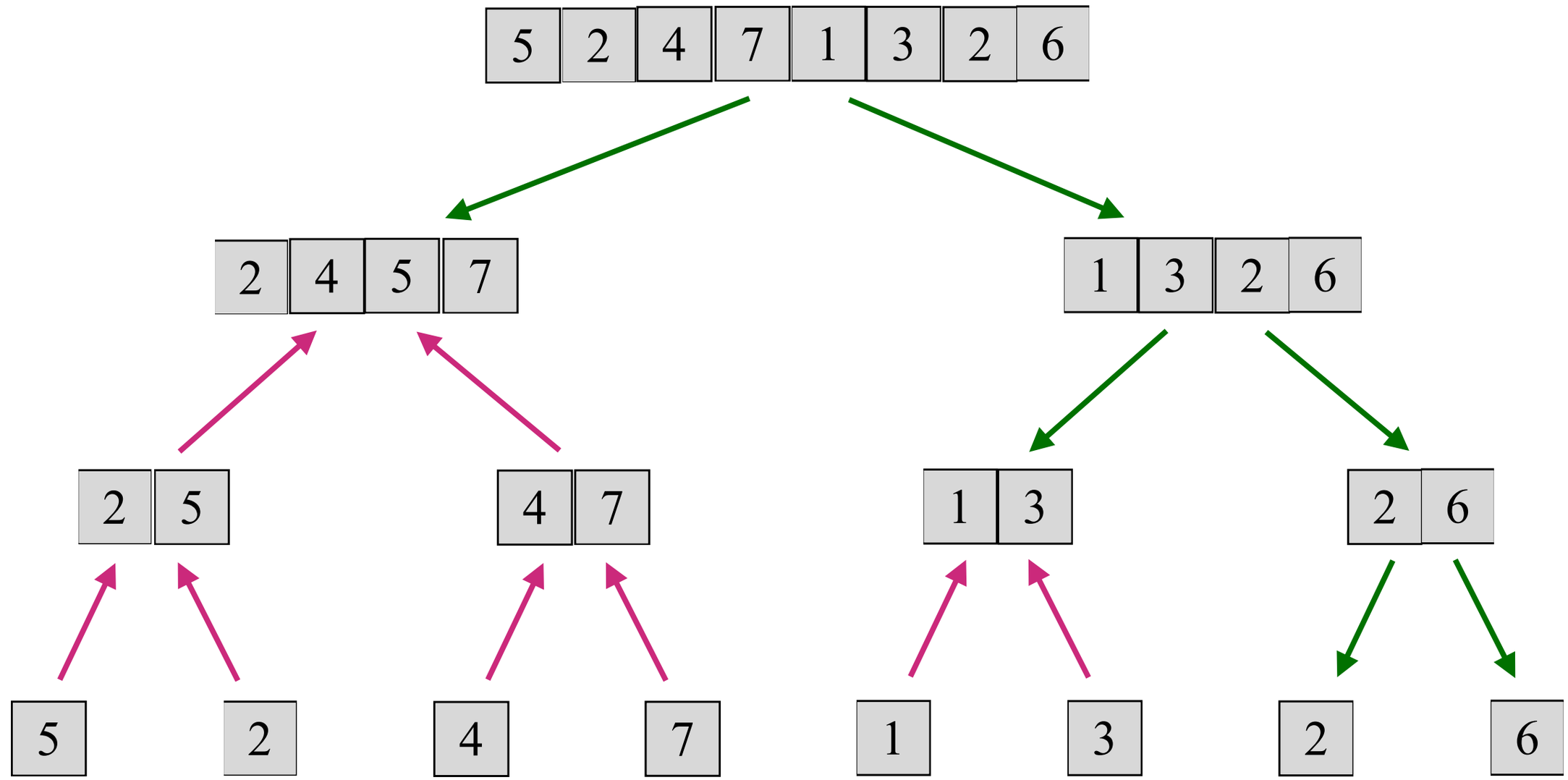
Merge Sort: Example



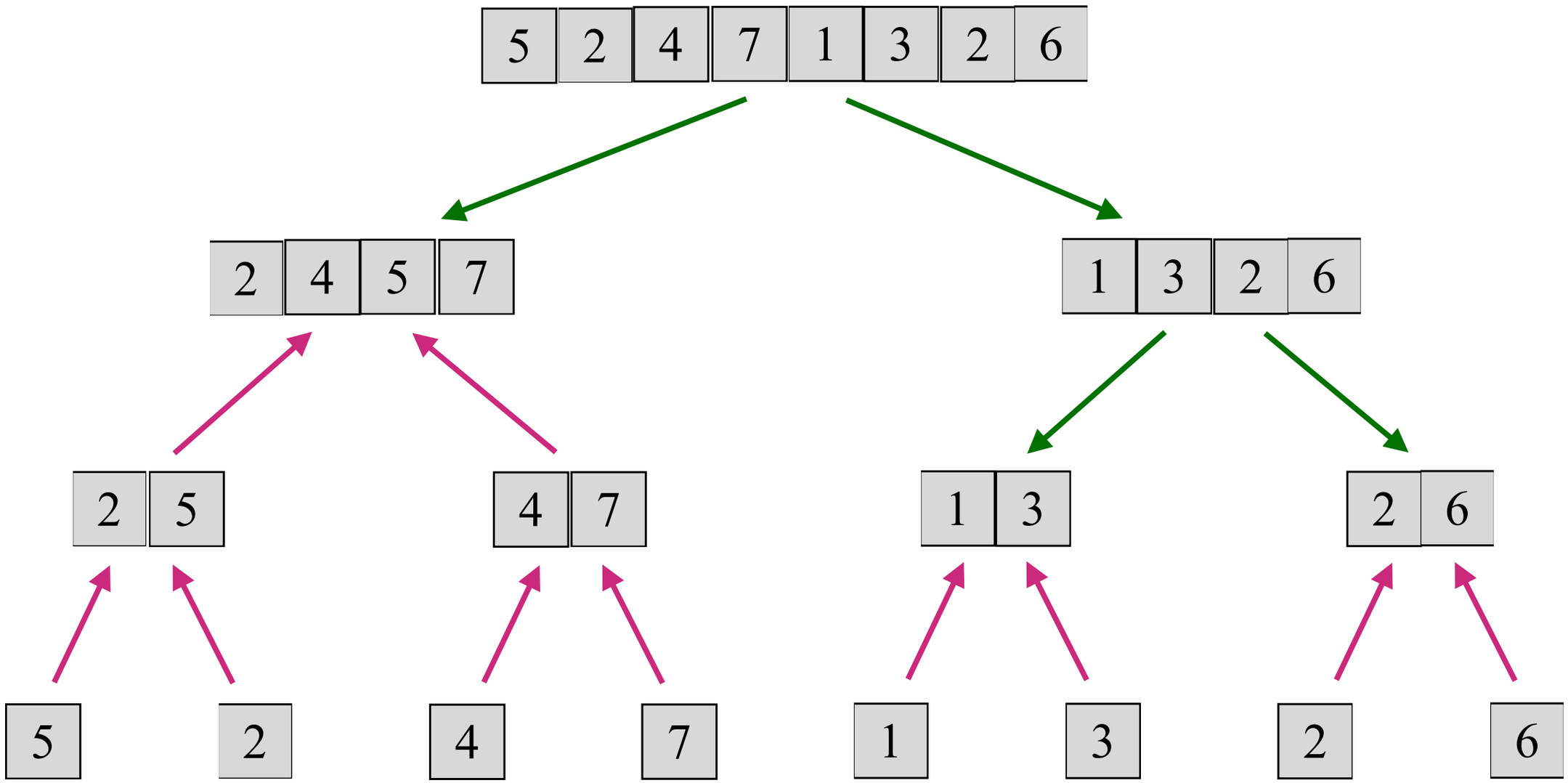
Merge Sort: Example



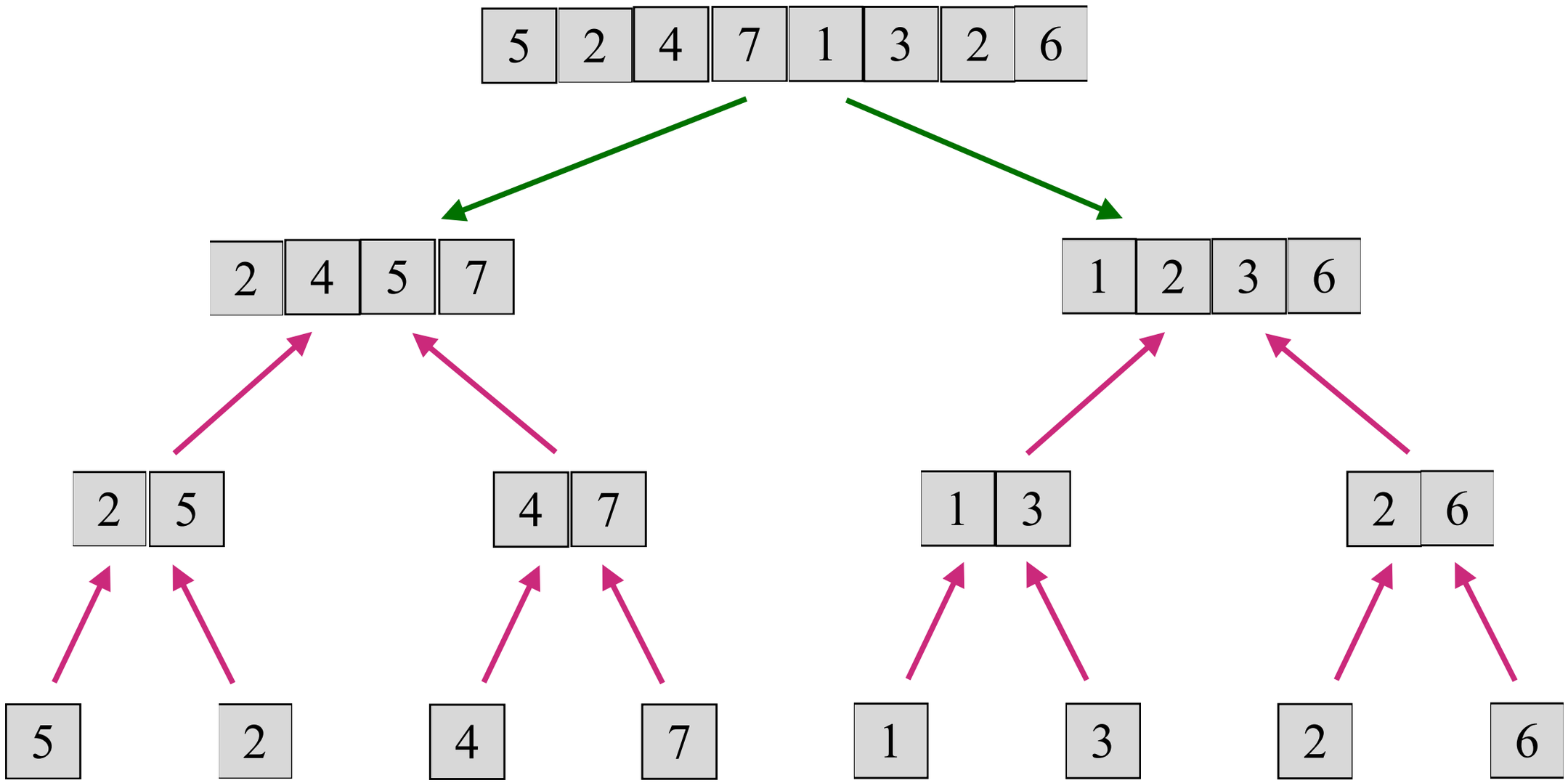
Merge Sort: Example



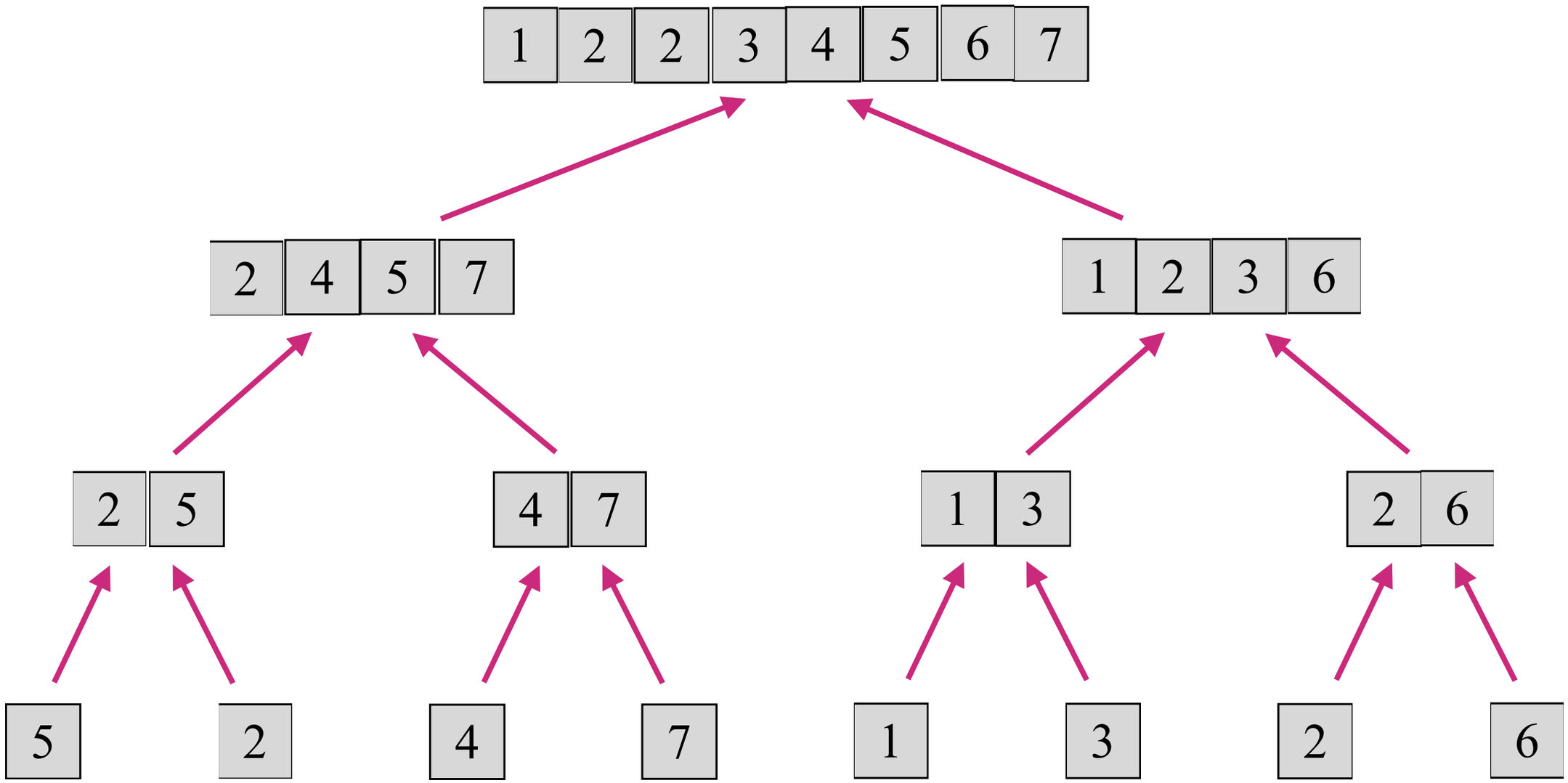
Merge Sort: Example



Merge Sort: Example



Merge Sort: Example



Merge sorted piles of cards

- **Input:** We have two sorted piles of cards face up on the table —one on the right (R) and one on the left (L)
- **Output:** we wish to merge the two decks into a single sorted pile, which is to be face down on the table

Main Idea:

- Repeat until one of the two piles is empty:
  - Chose the smaller of the two cards on top of the piles
  - Place the card face down onto the output pile
- Take the remaining input pile and place it face down onto the output pile

Merge

- It is the heart of the Merge-Sort algorithm
- **Input:** when we call MERGE(A, p, q, r) we assume that
  - $A[1..n]$ is an array and $1 \leq p \leq q < r \leq n$
    - The subarrays $A[p\,..\,q]$ and $A[q+1..r]$ are in sorted order
- **Output:** the subarray $A[p\,..\,r]$ is sorted, and the rest of the array is left untouched
- The procedure takes time $\Theta(n)$ where $n = r - p + 1$ is the total number of elements being merged

Merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

Merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

Initialise local data structures

Merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

Merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

Copy the content of the subarrays $A[p\,..\,q]$ and $A[q+1..r]$ respectively in L and R

Merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

Merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

To simplify the code.
We place at the bottom of the 'piles' L and R a sentinel card.

Merge

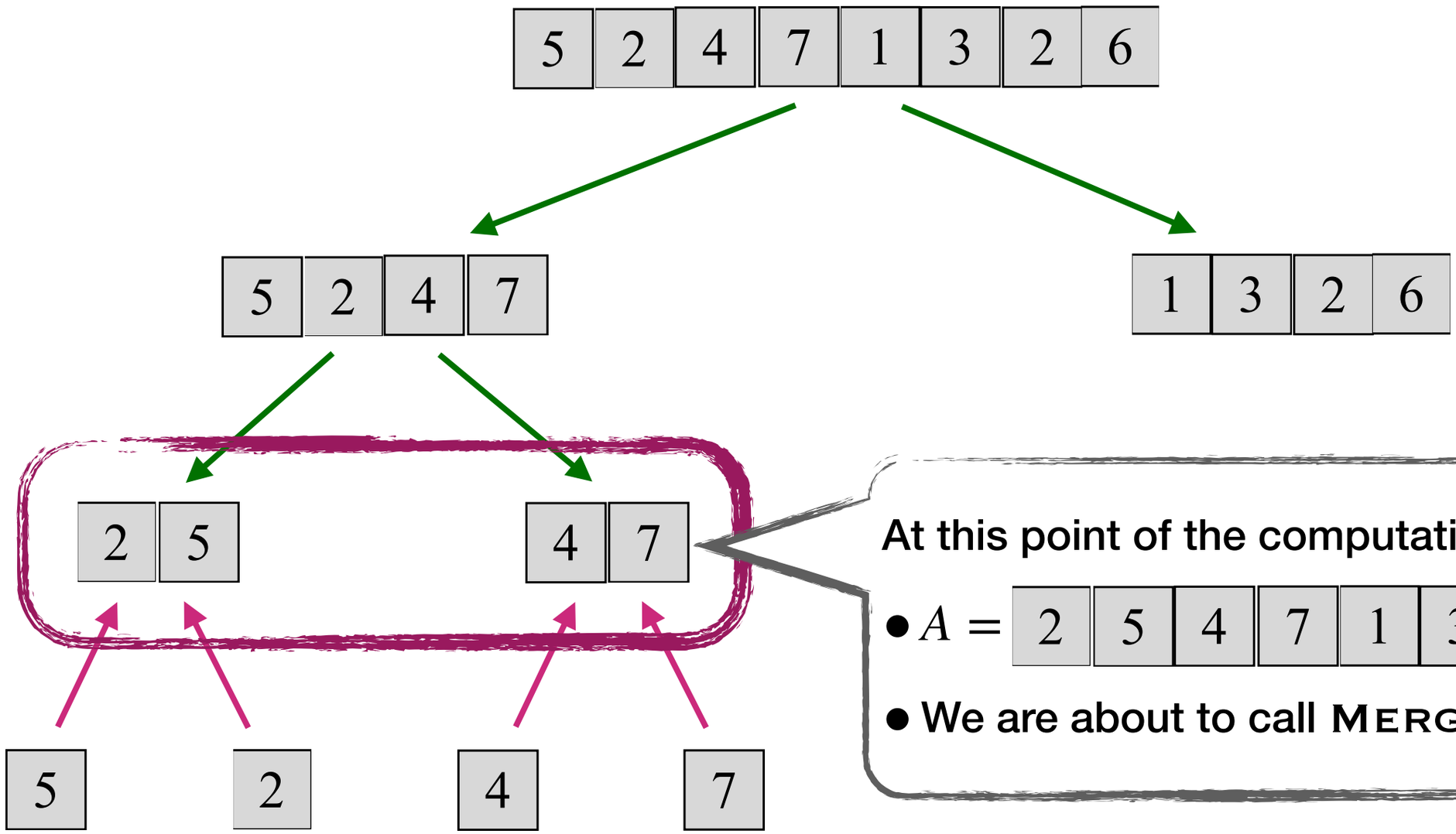
```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

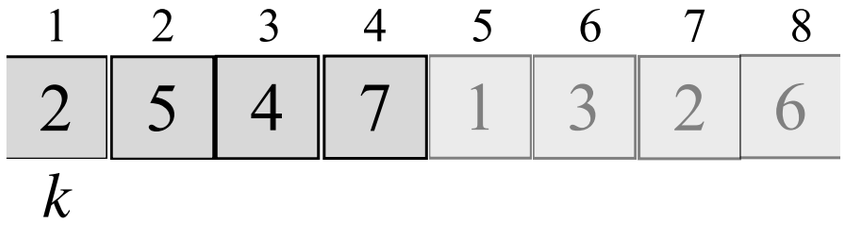
Iterate the procedure explained before:
chose the smallest element among $L[i]$ and $R[j]$ and copy that into $A[k]$

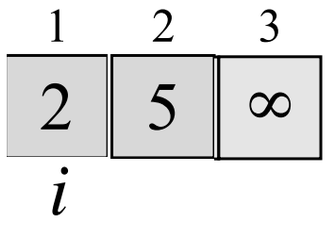
Merge: Example

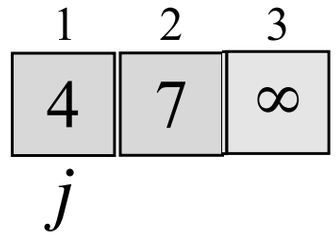


Merge: Example

$A[1..n]$ $L[1..n_1 + 1]$ $R[1..n_2 + 1]$

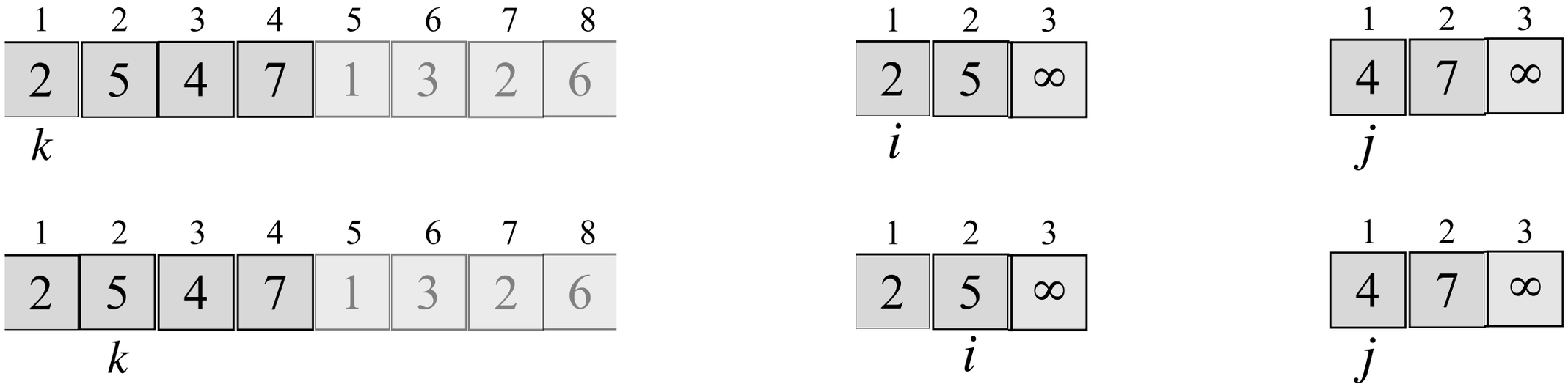






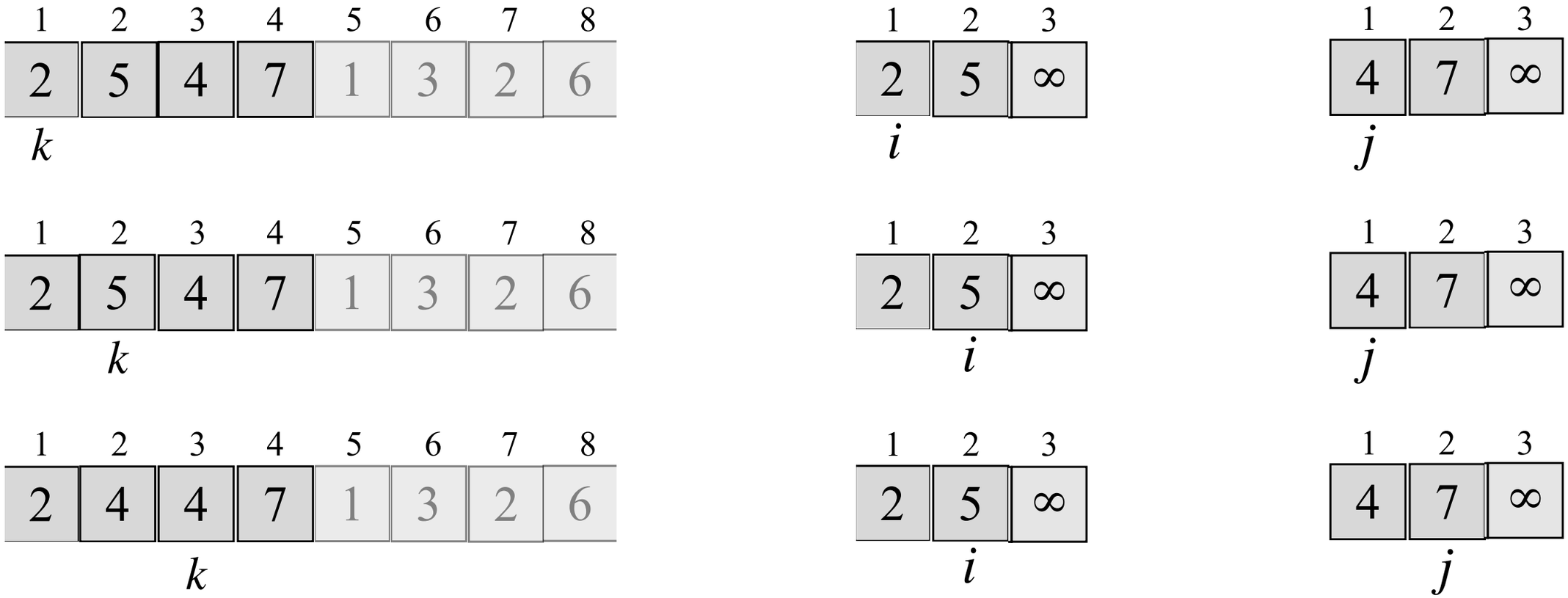
Merge: Example

$A[1..n]$ $L[1..n_1 + 1]$ $R[1..n_2 + 1]$



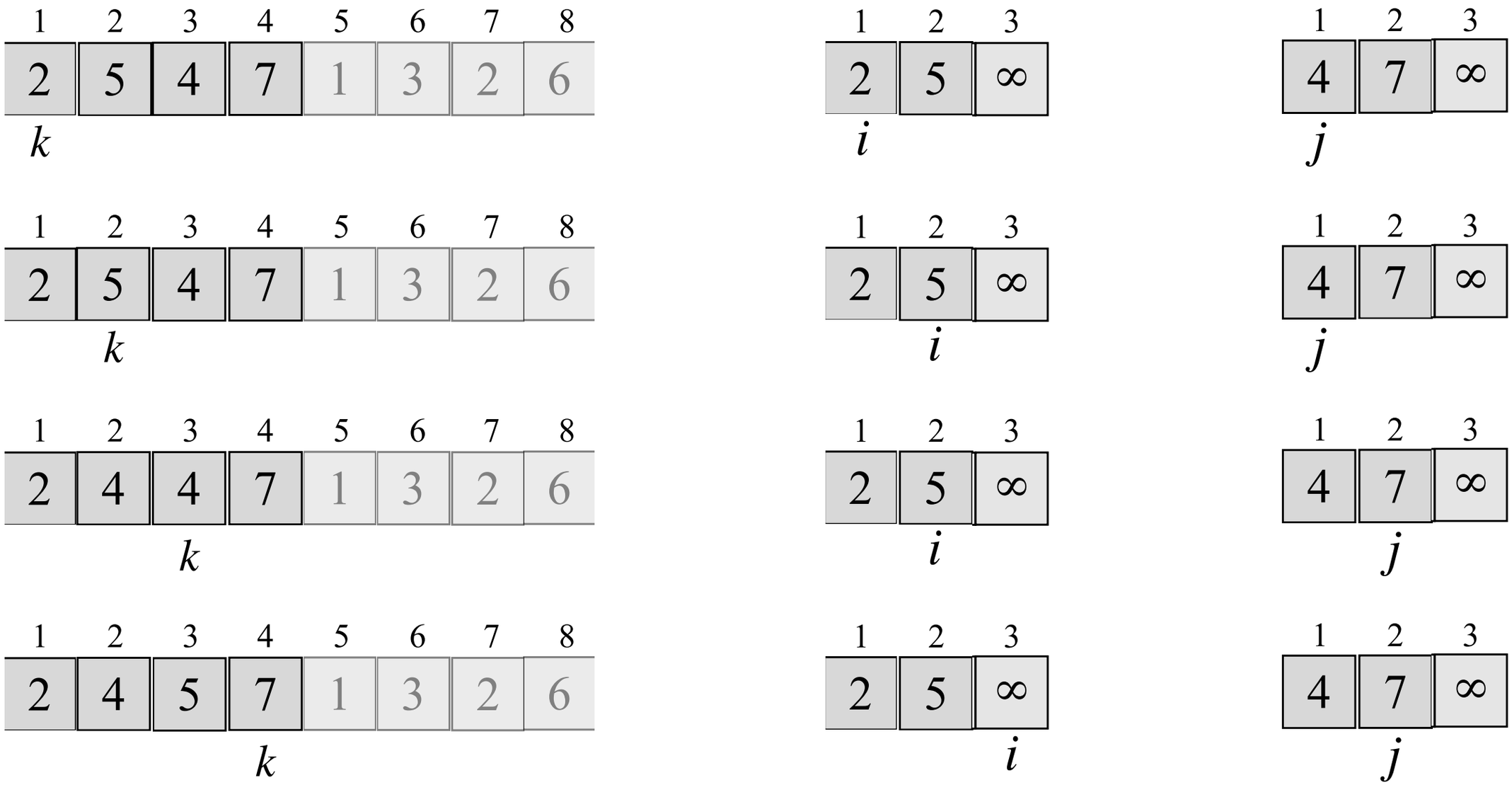
Merge: Example

$A[1..n]$ $L[1..n_1 + 1]$ $R[1..n_2 + 1]$



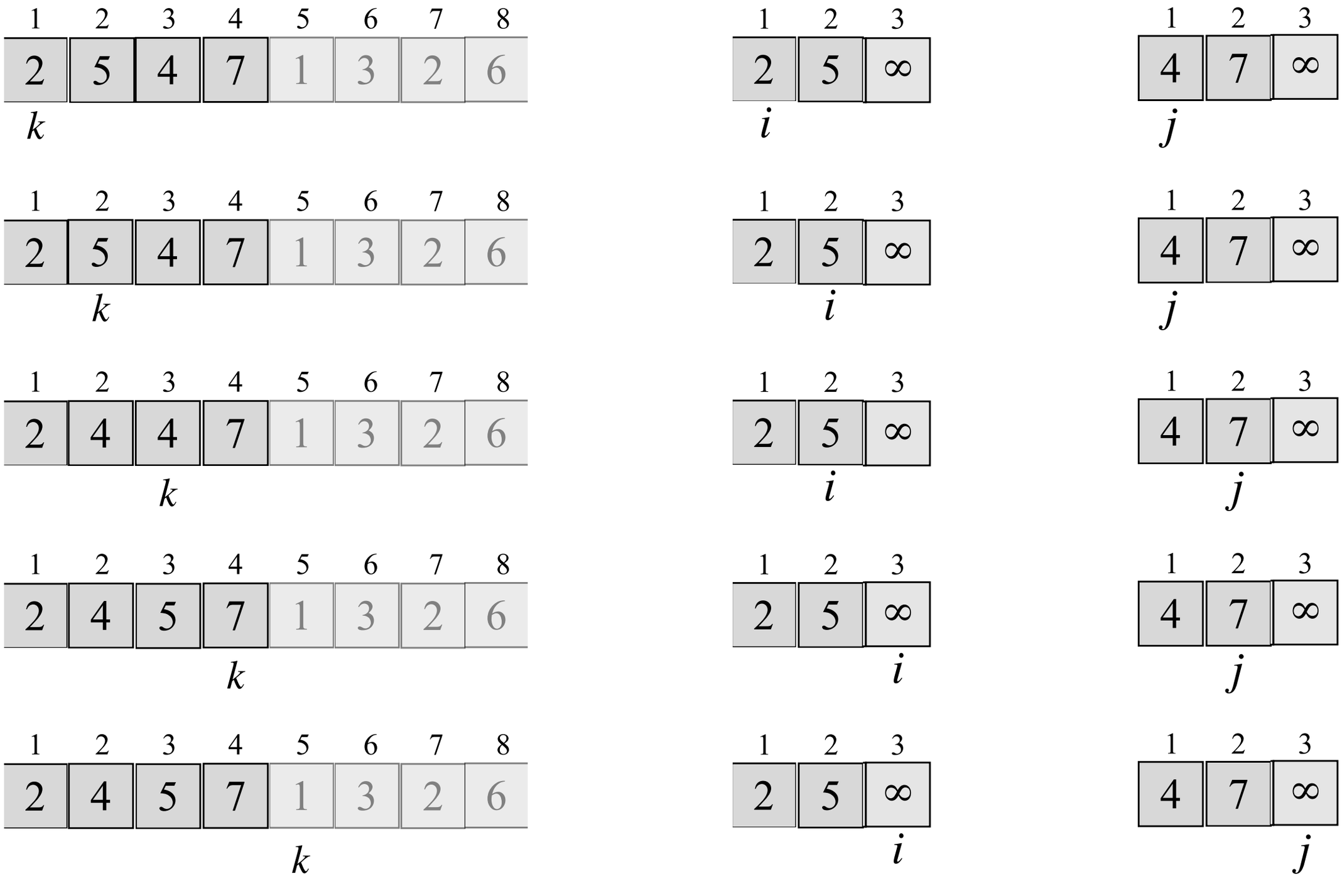
Merge: Example

$A[1..n]$ $L[1..n_1 + 1]$ $R[1..n_2 + 1]$



Merge: Example

$A[1..n]$ $L[1..n_1 + 1]$ $R[1..n_2 + 1]$



Merge: Correctness

We will only prove the correctness of the loop in line 10-17, assuming that the initialisation of L and R is performed as intended

Merge: Correctness

We will only prove the correctness of the loop in line 10-17, assuming that the initialisation of L and R is performed as intended

Loop Invariant

At the start of each iteration of the for loop, 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.

Merge: initialisation

Loop Invariant

At the start of each iteration of the for loop, 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.

We have to show that the loop invariant holds before the first loop iteration:

- Before the first loop iteration $k = p$, and $i = j = 1$.
- Therefore $A[p\,..\,k-1]$ is empty and contains the $k - p = 0$ smallest elements of L and R
- Since $i = j = 1$ and L and R are sorted, both $L[i]$ and $R[i]$ are the smallest elements of their arrays not been copied back into A.

Merge: maintenance

Loop Invariant

At the start of each iteration of the for loop, 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.

We have to show that each iteration maintains the invariant:

- We consider two cases (1) $L[i] \leq R[j]$ and (2) $L[i] > R[j]$:

1. If $L[i] \leq R[j]$, then $L[i]$ is the smallest element not yet copied back into A. Because $A[p \,..\, k-1]$ contains the $k-p$ smallest elements, after copying $L[i]$ into $A[k]$, the subarray $A[p \,..\, k]$ constants the $k-p+1$ smallest elements. Incrementing k (in the for loop update) and i (in line 15) restore the loop invariant for the next iteration.
2. If $L[i] > R[j]$, we can proceed using similar arguments but now we replace L and i, respectively with R and j.

Merge: termination

Loop Invariant

At the start of each iteration of the for loop, 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.

- At termination $k = r+1$
- By the loop invariant, the subarray $A[p \,..\, k-1] = A[p \,..\, r]$, contains the $k-p = r-p+1$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order.
- L and R together contain $n_1+n_2+2 = r-p+3$ elements.
- Thus all but the two largest have been copied back into A, and these two largest are the sentinels ∞.

Merge: Runtime Analysis

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

$$T(n) = \Theta(n)$$

Merge: Runtime Analysis

A meaningful size of an instance is $n = r - p + 1$, i.e., the number elements to merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

$$T(n) = \Theta(n)$$

Merge: Runtime Analysis

A meaningful size of an instance is $n = r - p + 1$, i.e., the number elements to merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
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
```

$\Theta(1)$

$$T(n) = \Theta(n)$$

Merge: Runtime Analysis

A meaningful size of an instance is $n = r - p + 1$, i.e., the number elements to merge

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1                                          Θ(1)
 2  n_2 = r − q                                              Θ(1)
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays        Θ(1)
 4  for i = 1 to n_1                                         Θ(n_1)
 5      L[i] = A[p + i − 1]                                  Θ(n_1)
 6  for j = 1 to n_2                                         Θ(n_2)
 7      R[j] = A[q + j]                                      Θ(n_2)
 8  L[n_1 + 1] = ∞                                           Θ(1)
 9  R[n_2 + 1] = ∞                                           Θ(1)
10  i = 1                                                    Θ(1)
11  j = 1                                                    Θ(1)
12  for k = p to r                                           Θ(n_1 + n_2) = Θ(n)
13      if L[i] ≤ R[j]                                       Θ(n)
14          A[k] = L[i]                                      Θ(n)
15          i = i + 1                                        Θ(n)
16      else A[k] = R[j]                                     Θ(n)
17          j = j + 1                                        Θ(n)
```

$$T(n) = \Theta(n)$$

Merge: Runtime Analysis

A meaningful size of an instance is $n = r - p + 1$, i.e., the number elements to merge

```
MERGE(A, p, q, r)
1   n_1 = q − p + 1                                                    Θ(1)
2   n_2 = r − q
3   let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4   for i = 1 to n_1
5       L[i] = A[p + i − 1]                                            Θ(n_1 + n_2) = Θ(n)
6   for j = 1 to n_2
7       R[j] = A[q + j]
8   L[n_1 + 1] = ∞
9   R[n_2 + 1] = ∞                                                     Θ(1)
10  i = 1
11  j = 1
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
```

$$T(n) = \Theta(n)$$

Merge: Runtime Analysis

A meaningful size of an instance is $n = r - p + 1$, i.e., the number elements to merge

```
MERGE(A, p, q, r)
1   n_1 = q − p + 1                                              Θ(1)
2   n_2 = r − q                                                  Θ(1)
3   let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays            Θ(n_1 + n_2) = Θ(n)
4   for i = 1 to n_1                                             Θ(n_1) = Θ(n)
5       L[i] = A[p + i − 1]                                      Θ(1)
6   for j = 1 to n_2                                             Θ(n_2) = Θ(n)
7       R[j] = A[q + j]                                          Θ(1)
8   L[n_1 + 1] = ∞                                               Θ(1)
9   R[n_2 + 1] = ∞                                               Θ(1)
10  i = 1                                                        Θ(1)
11  j = 1                                                        Θ(1)
12  for k = p to r                                               Θ(r − p) = Θ(n)
13      if L[i] ≤ R[j]                                           Θ(1)
14          A[k] = L[i]                                          Θ(1)
15          i = i + 1                                            Θ(1)
16      else A[k] = R[j]                                         Θ(1)
17          j = j + 1                                            Θ(1)
```

$$T(n) = \Theta(n)$$

Merge Sort: Correctness

We prove that after a call to $\textsc{Merge}(A, p, r)$ the subarray $A[p\,..\,r]$ is sorted.

We proceed by induction on $n = r - p$

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

- **Base Case** ($n = 0$): Then, $r = p$ and the subarray $A[r]$ is trivially sorted.
- **Inductive step** ($n > 0$): Then, $p < r$, $q - p < n$ and $r - q - 1 < n$ hold.
  - By inductive hypothesis after calling $\textsc{Merge-Sort}(A, p, q)$ and $\textsc{Merge-Sort}(A, q + 1, r)$ the subarrays $A[p\,..\,q]$ and $A[q + 1..r]$ are sorted.
  - Thus the $\textsc{Merge}$ procedure runs under the required conditions and correctly merge the two adjacent subarrays. Hence $A[p\,..\,r]$ is sorted.

Merge Sort: Running time

To simplify our analysis we assume that the problem size $n = r - p$ is a power of 2. Each divide steps then yields two subsequences of size exactly $n/2$.

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

- **Divide:** computing the middle of the subarray takes $\Theta(1)$
- **Conquer:** solving recursively two subproblems each of size $n/2$, contributes $2T(n/2)$ to the running time
- **Combine:** the merge takes $\Theta(n)$ on an n-elements subarray.

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

How to solve the recurrence

Let's rewrite the recurrence as

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

where c represents the time to solve problems of size 1 as well as the time per array element of the divide and combine steps*

How to solve the recurrence

Let’s rewrite the recurrence as

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

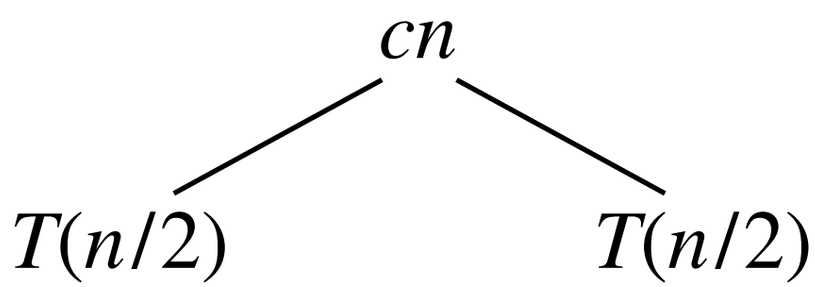
$$T(n)$$

How to solve the recurrence

Let's rewrite the recurrence as

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

where c represents the time to solve problems of size 1 as well as the time per array element of the divide and combine steps*

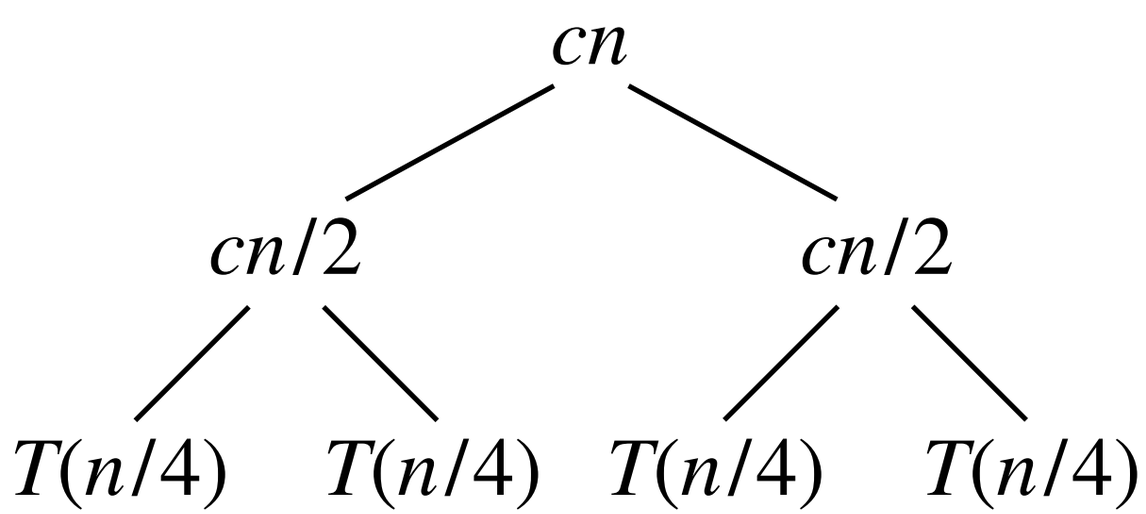


How to solve the recurrence

Let's rewrite the recurrence as

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

where c represents the time to solve problems of size 1 as well as the time per array element of the divide and combine steps*

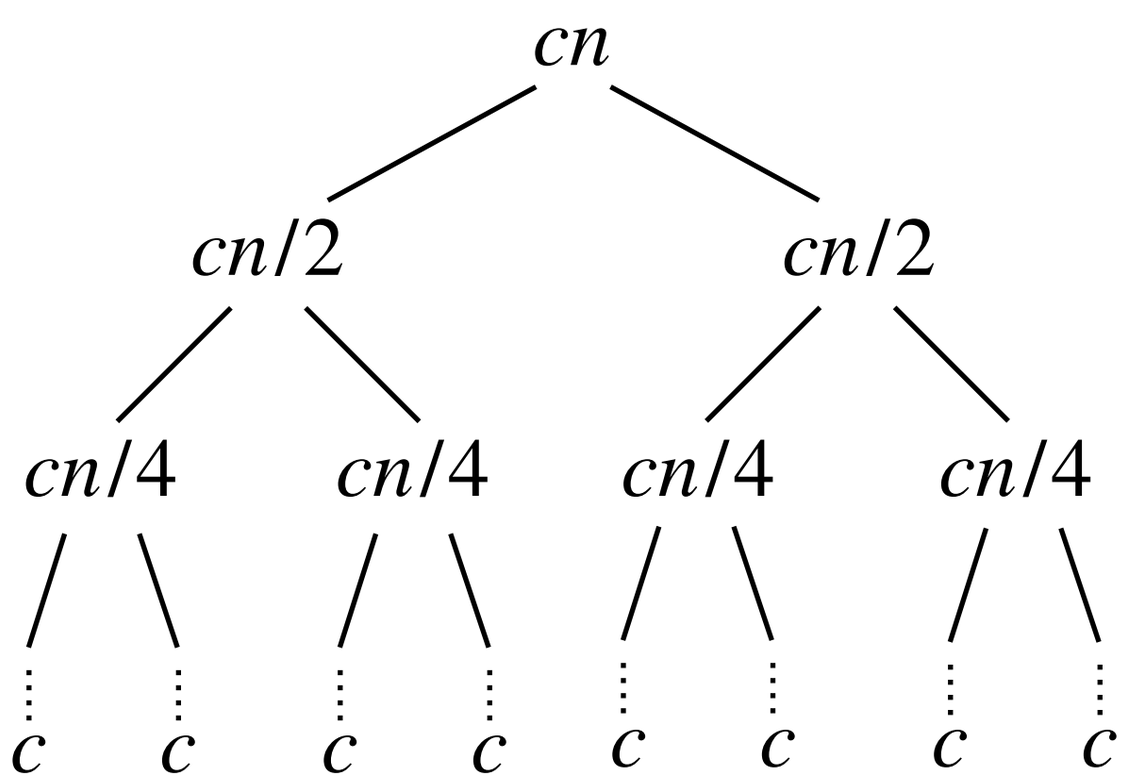


How to solve the recurrence

Let's rewrite the recurrence as

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

where c represents the time to solve problems of size 1 as well as the time per array element of the divide and combine steps*

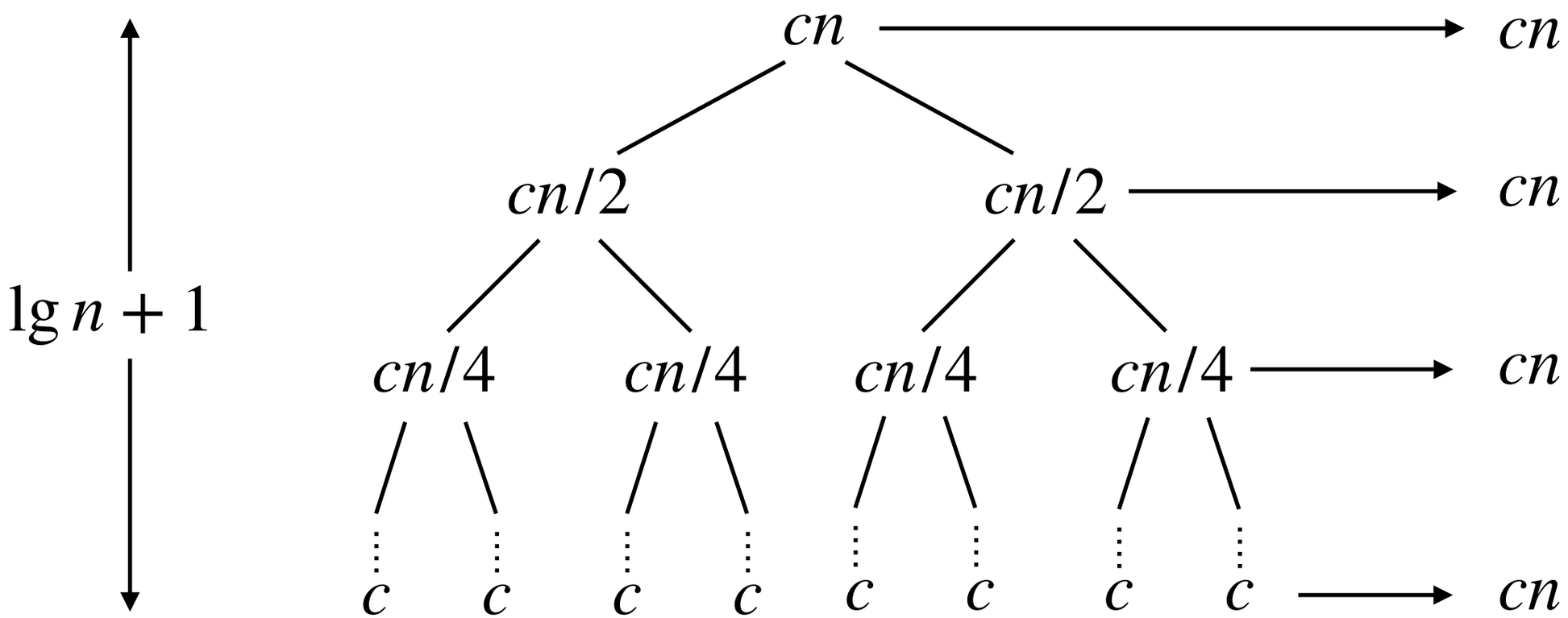


How to solve the recurrence

Let's rewrite the recurrence as

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

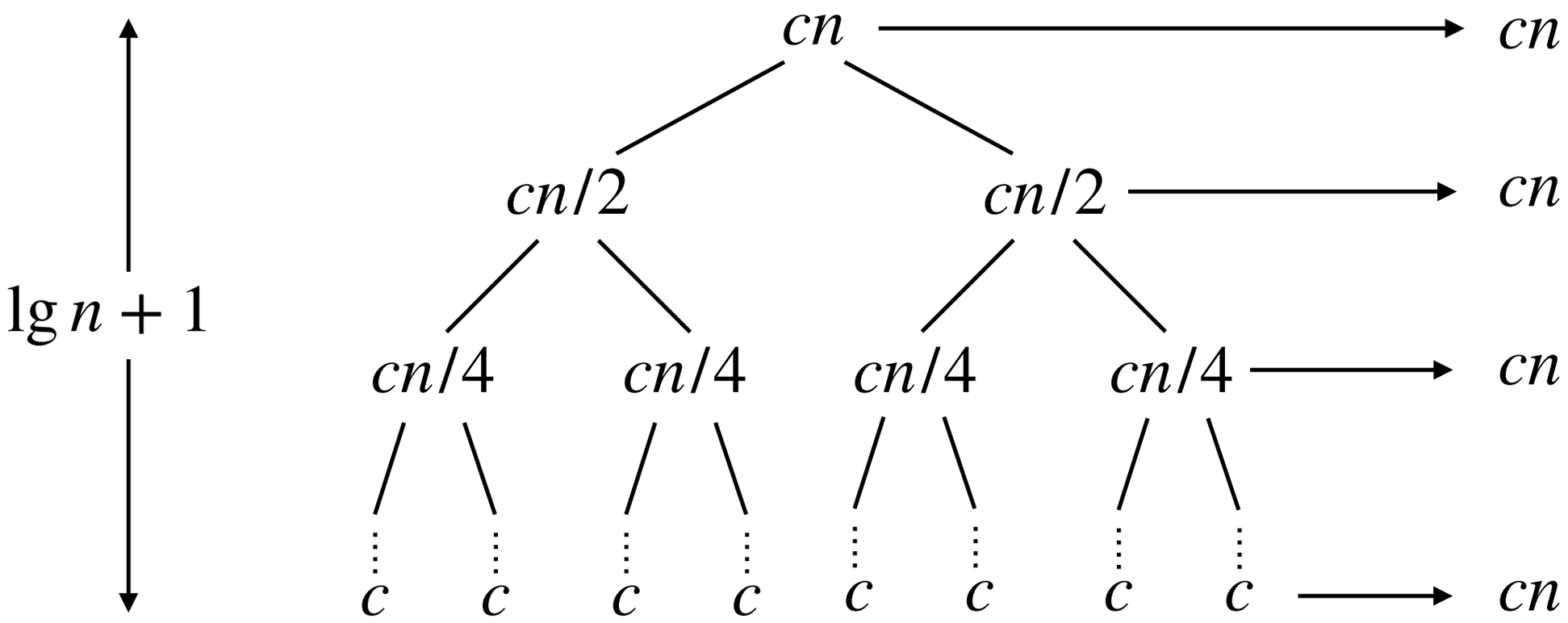
where c represents the time to solve problems of size 1 as well as the time per array element of the divide and combine steps*



How to solve the recurrence

Therefore in total we have

$$T(n) = cn \lg n + cn = \Theta(n \lg n)$$



A General Framework

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

A General Framework

Size for which the problem becomes trivial to solve

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

A General Framework

Size for which the problem becomes trivial to solve

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

Division of the problem $b > 1$

A General Framework

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

Size for which the problem becomes trivial to solve

Number of recursive calls to the subproblems

Division of the problem $b > 1$

A General Framework

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

Size for which the problem becomes trivial to solve

Number of recursive calls to the subproblems

Division of the problem $b > 1$

Running time for the divide step

A General Framework

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

Size for which the problem becomes trivial to solve

Number of recursive calls to the subproblems

Division of the problem $b > 1$

Running time for the divide step

Running time for the combine step

Quiz: Binary Search

Assume that $A[1..n]$ where n is a power of 2

- What is the recurrence for the Bin-Search algorithm?
- Can we use the recursion tree method in this case to solve the recurrence?

```
BIN-SEARCH(A, l, r, a)
 1  if l < r
 2        m = ⌊(l + r)/2⌋
 3        if A[m] ≥ a
 4              return BIN-SEARCH(A, l, m, a)
 5        else
 6              return BIN-SEARCH(A, m + 1, r, a)
 7  elseif l = r and A[l] = a
 8        return l
 9  else
10        return 0
```

Learned Today

- Divide and Conquer
  - Effective algorithmic design principle
  - Efficient when subproblems are disjoint

- Merge-Sort algorithm
  - Worst-case running time $\Theta(n \lg n)$
  - Uses $\Theta(n)$ additional memory

- Analysis Techniques:
  - Correctness of recurrences using **induction**
  - Runtime analysis using the **recursion tree method**