Algorithms
Chapter 2 Getting Started

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Insertion sort**
- Analyzing algorithms
- Designing algorithms

The purpose of this chapter

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- Learn how to prove the correctness of an algorithm.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of “divide and conquer” in the context of merge sort.

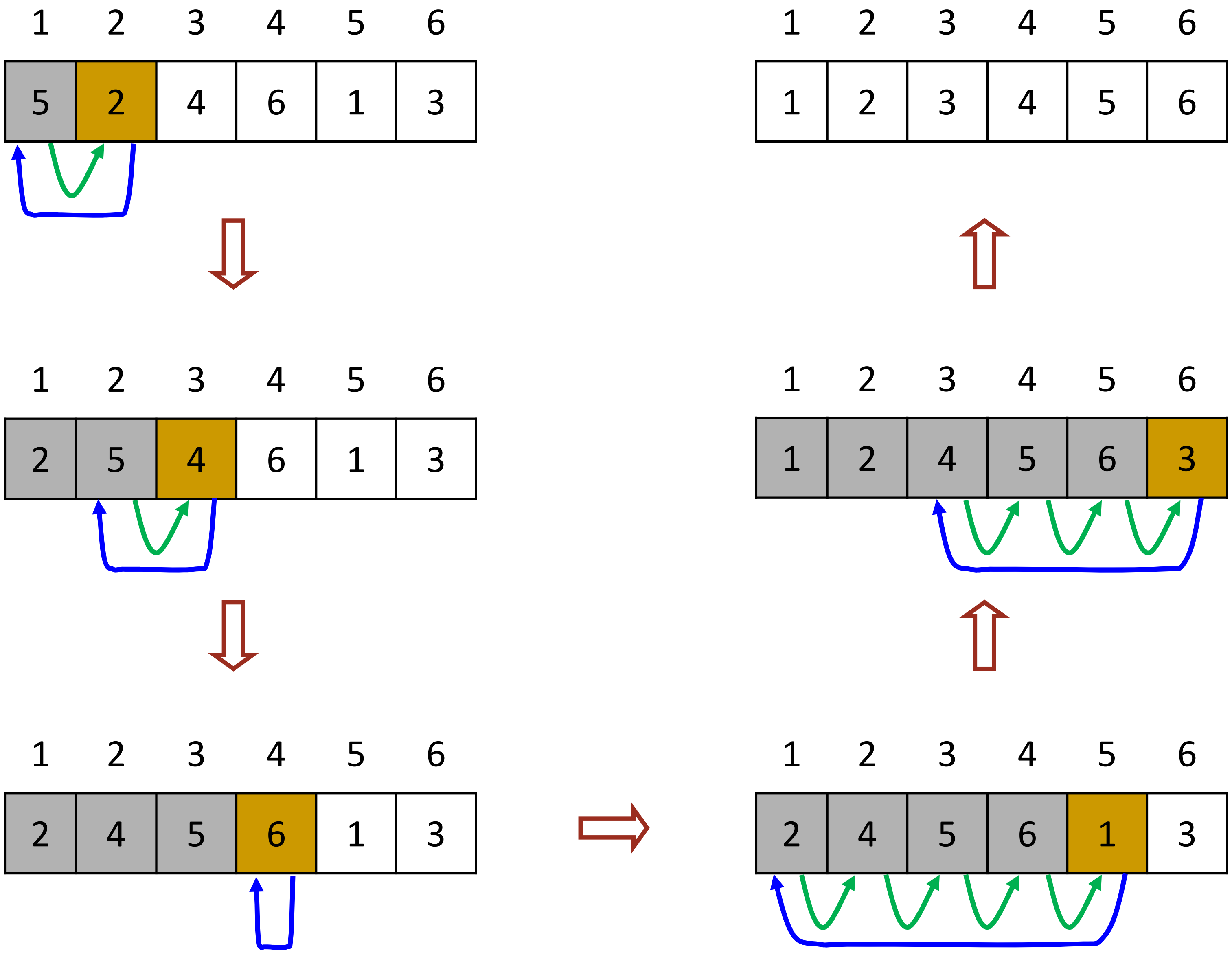
Algorithm

- **Algorithm**: a well-defined computational procedure that takes some value as input and produces some value as output.
- Major concerns:
  - Correctness
  - Time complexity
- For example: The sorting problem
  - **Input**: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Output**: A permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$.
  - Given the input sequence 31, 41, 59, 26, 41, 58, a sorting algorithm returns as output the sequence 26, 31, 41, 41, 58, 59.

Insertion sort

- Insertion sort: an efficient algorithm for sorting a small number of elements.

```
INSERTION-SORT(A)
    for j ← 2 to length[A]
        do key ← A[j]
            /* Insert A[j] into the sorted sequence A[1… j − 1]*/
            i ← j − 1
            while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                    i ← i − 1
            A[i + 1] ← key
```


1 2 3 4 5 6
5 2 4 6 1 3
1 2 3 4 5 6
2 5 4 6 1 3
1 2 3 4 5 6
2 4 5 6 1 3
1 2 3 4 5 6
2 4 5 6 1 3
1 2 3 4 5 6
1 2 4 5 6 3
1 2 3 4 5 6
1 2 3 4 5 6

Loop invariant for proving correctness

- We may use **loop invariants** to prove the correctness.
  - **Initialization**: It is true before the first iteration of the loop.
  - **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
  - **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.
- Using loop invariants is like mathematical induction.

Correctness of INSERTION-SORT

- **Loop invariant**: At the start of each iteration of the **for** loop of lines 1-8, the subarray $A[1...j-1]$ consists of the elements originally in $A[1...j-1]$ but in sorted order.
  - **Initialization**: Before the first iteration, $j = 2$. $A[1]$ is trivially sorted.
  - **Maintenance**: Note that the body of the outer **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, ..., and so on by one position to the right until the proper position for $A[j]$ is found.
  - **Termination**: The outer **for** loop ends when j exceeds n, i.e., when $j = n + 1$. Then, $A[1...n]$ is sorted.

Outline

- Insertion sort
- **Analyzing algorithms**
- Designing algorithms

Time complexity of INSERTION-SORT

- Let t_j be the number of times the while loop test for value j.

INSERTION-SORT(A)	cost	times
for $j \leftarrow 2$ **to** *length*[A]	c_1	n
do *key* $\leftarrow A[j]$	c_2	$n-1$
/* Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$. */	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

- $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Time complexity of INSERTION-SORT

- **Best-case**: The array is already sorted.
  - $t_2 = t_3 \ldots. = t_n = 1.$
  - $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
    $$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$
  - A **linear function** of n.
- **Worst-case**: The array is in reverse sorted order.
  - $t_2 = 2, t_3 = 3, \ldots., t_n = n.$
  - $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$
    $$+ c_6 \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1)$$
    $$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n$$
    $$- (c_2 + c_4 + c_5 + c_8)$$
  - A **quadratic function** of n.

Worst-case and average-case analysis

- We shall usually concentrate on finding only the worst-case
  - The worst-case running time gives us a guarantee that the algorithm will never take any longer.
  - For some algorithms, the worst case occurs fairly often.
  - The "average case" is often roughly as bad as the worst case.
- For example:
  - Consider the insertion sort, on average, we check half of the subarray $A[1\ldots j-1]$, so $t_j = j/2$.
  - The average-case running time is still a quadratic function of n.

Order of growth

- Another abstraction to ease analysis and focus on the important features.
- Look only at the leading term of the formula for running time.
  - Drop lower-order terms.
  - Ignore the constant coefficient in the leading term.
- For example:
  - The worst-case running time of insertion sort is $an^2 + bn + c$.
  - Drop lower-order terms $\Rightarrow an^2$.
  - Ignore constant coefficient $\Rightarrow n^2$.
  - We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2.

Outline

- Insertion sort
- Analyzing algorithms
- **Designing algorithms**

Designing algorithms

- There are many ways to design algorithms.
- **Incremental**:
  - For example of insertion sort, having sorted subarray $A[1\ldots j-1]$ and then yielding the sorted array $A[1\ldots j]$.
- **Divide and conquer**
  - **Divide** the problem into a number of subproblems.
  - **Conquer** the subproblems by solving them recursively.
    - If the subproblems sizes are small enough, just solve them in a straightforward manner.
  - **Combine** the subproblem solutions to give a solution to the original problem.

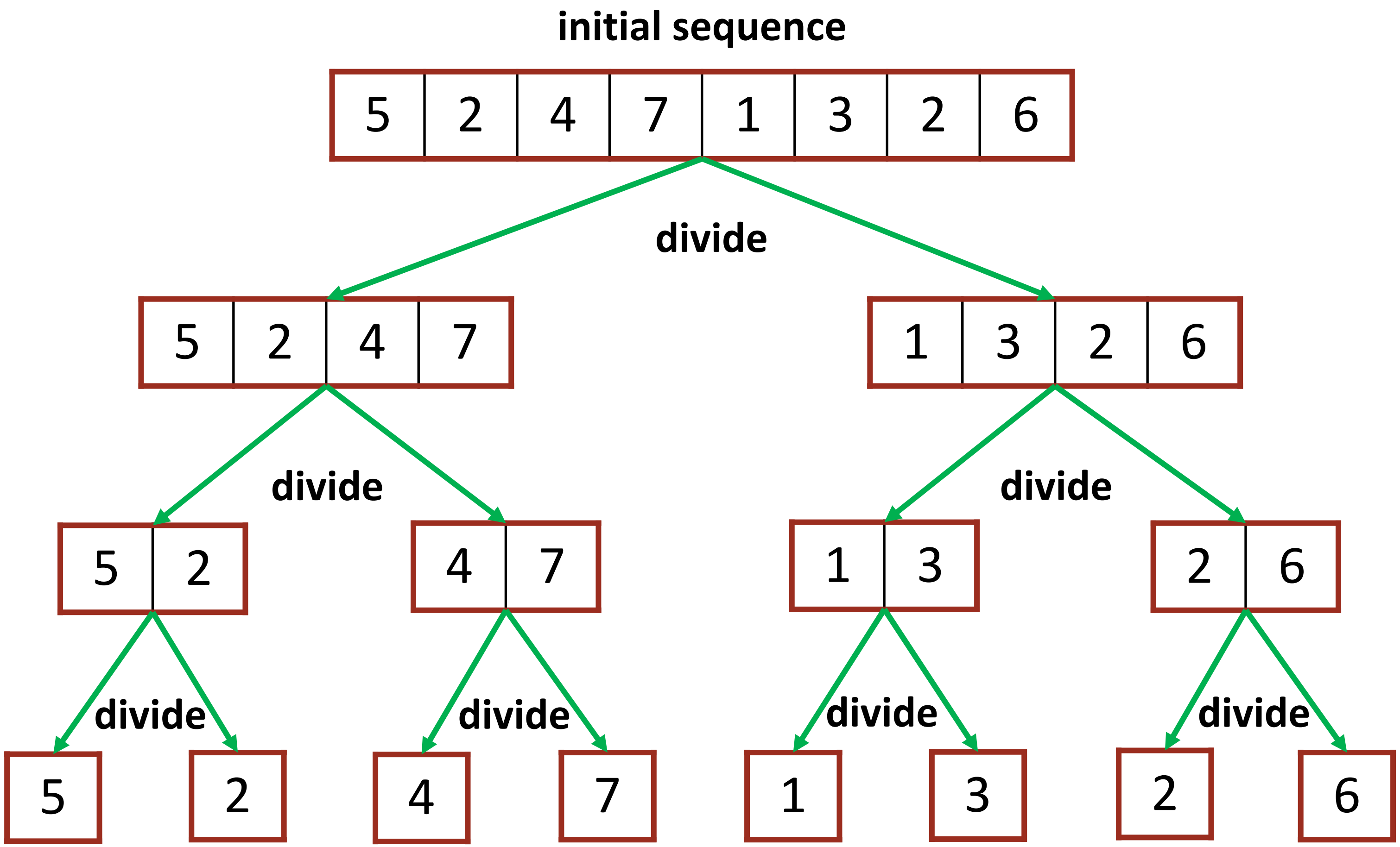
Merge sort

- **Divide** by splitting into two subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$, where q is the halfway point of $A[p \ldots r]$.
- **Conquer** by recursively sorting the two subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$.
- **Combine** by merging the two sorted subsequences to produce the sorted answer.

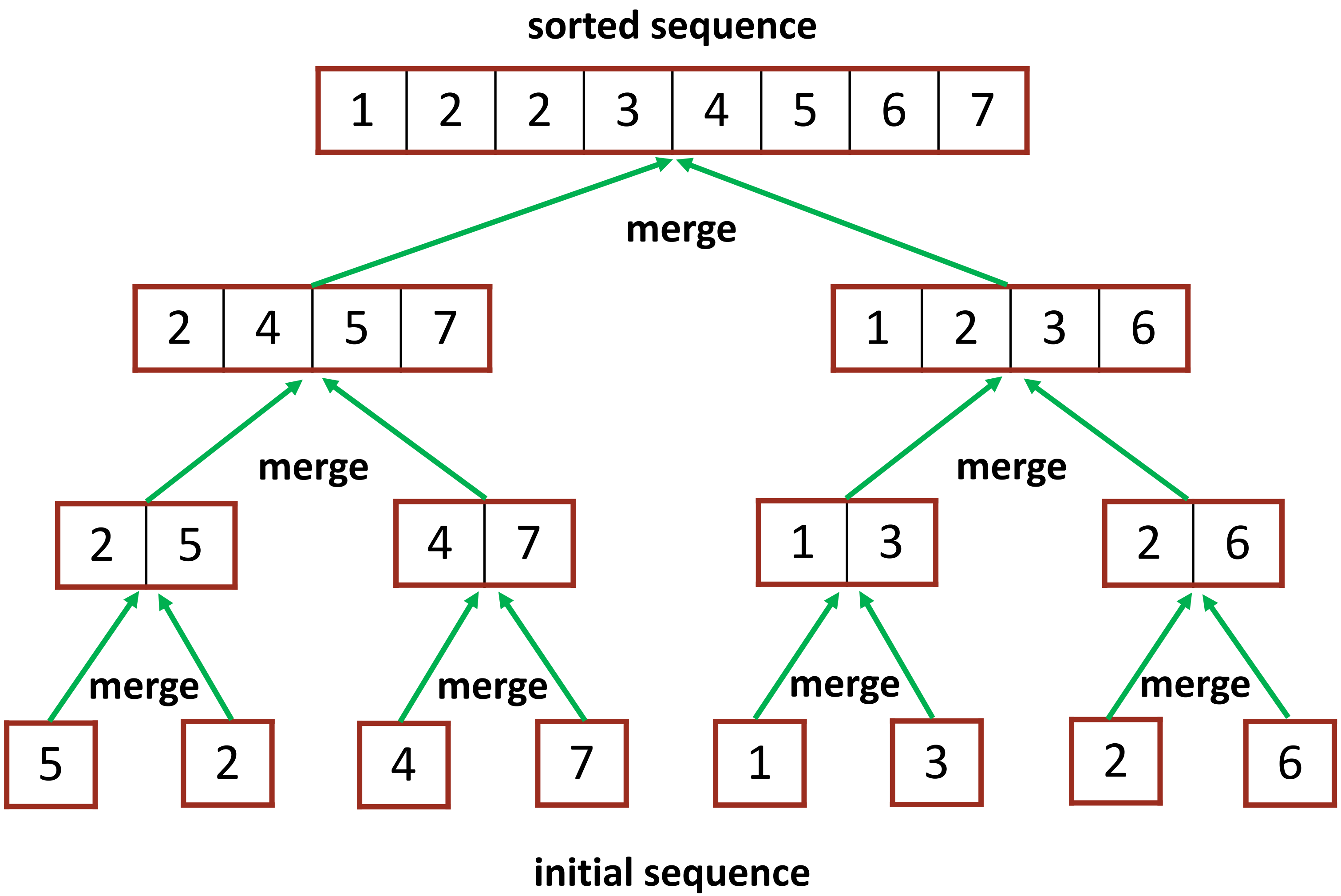
```
MERGE-SORT (A, p, r)
    if p < r                          //Check for base case
      then q ← ⌊(p+r)/2⌋              //Divide
           MERGE-SORT(A, p, q)        //Conquer
           MERGE-SORT(A, q+1, r)      //Conquer
           MERGE(A, p, q, r)          //Combine
```

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

An example for MERGE-SORT

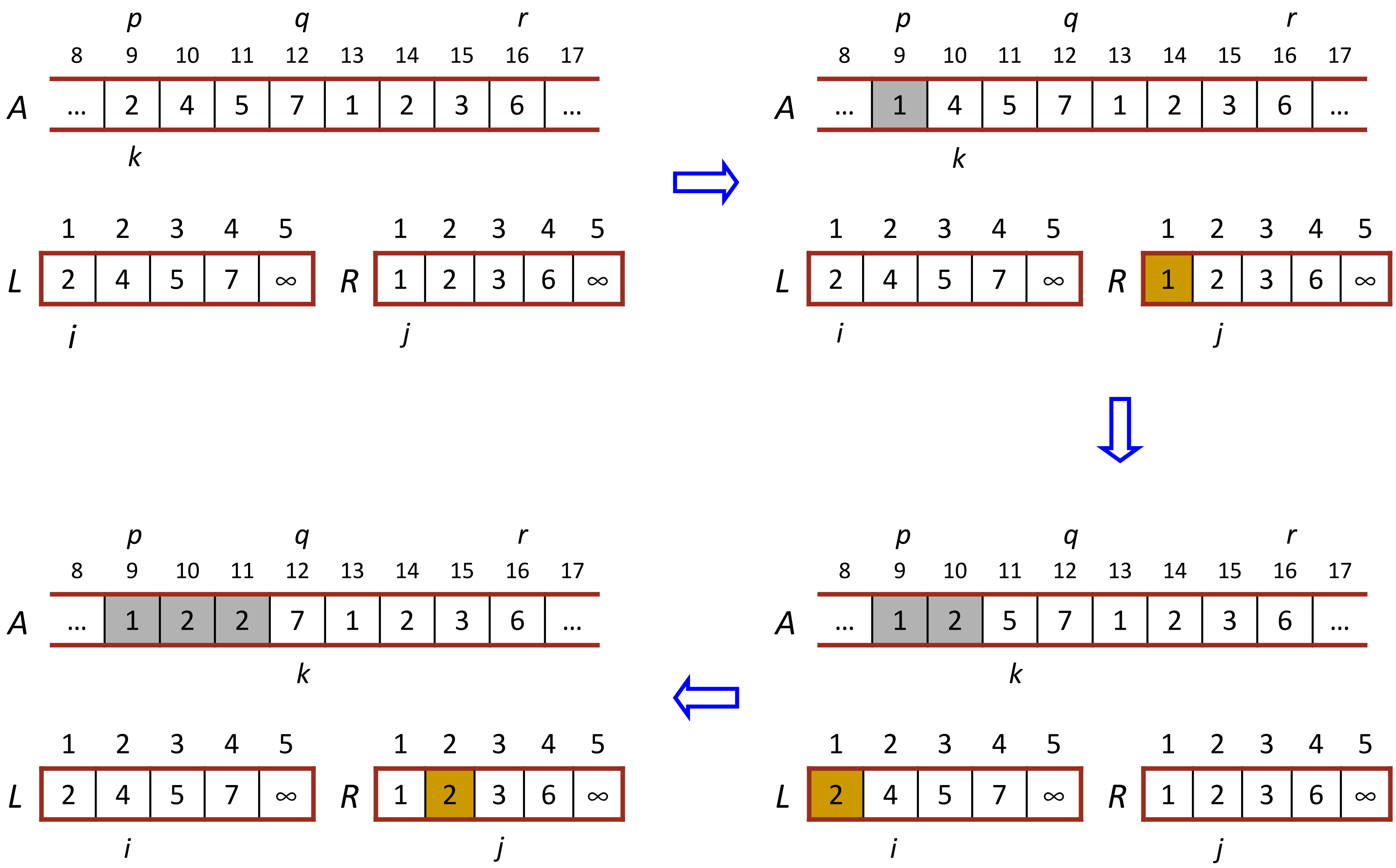


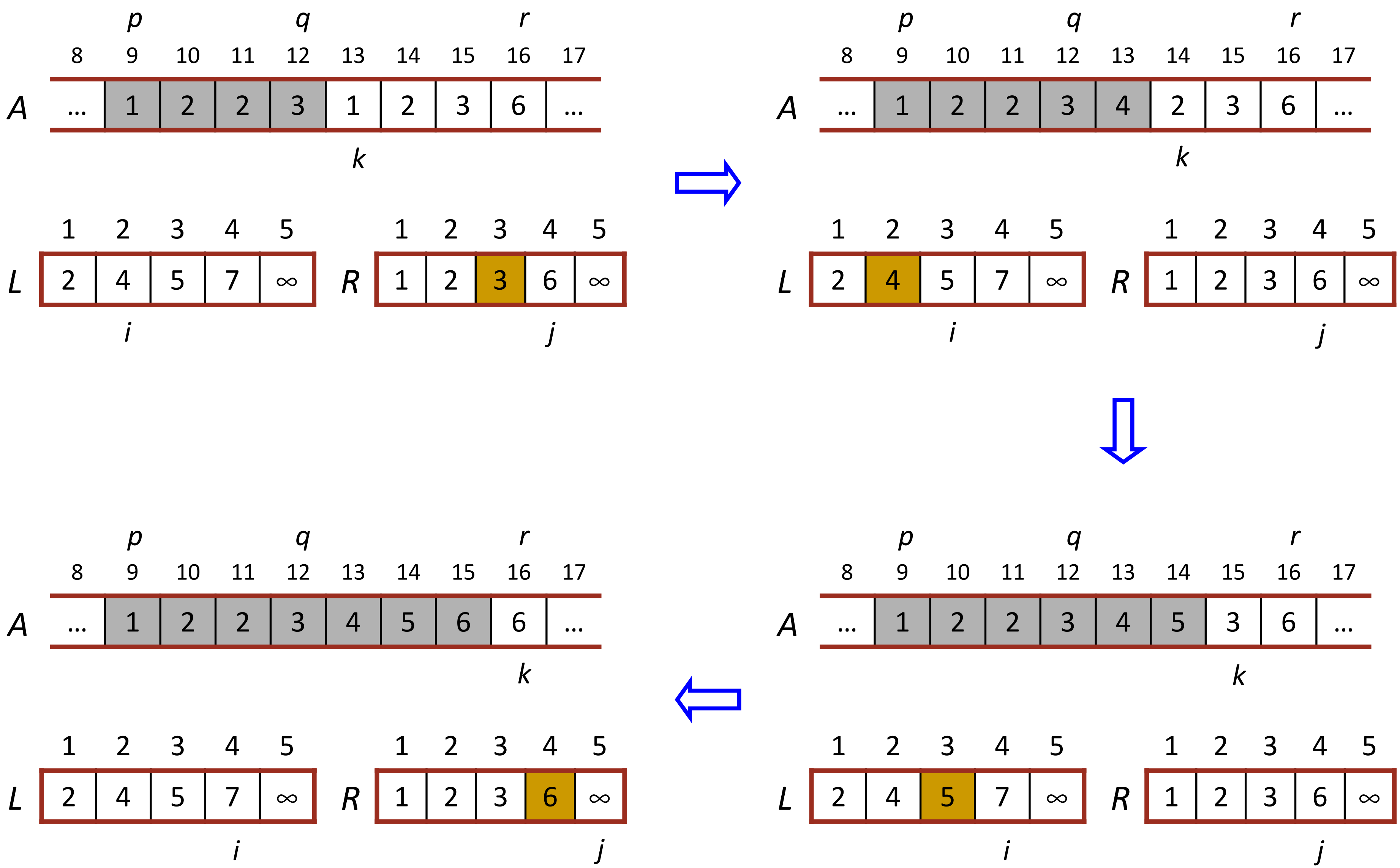
An example for MERGE-SORT

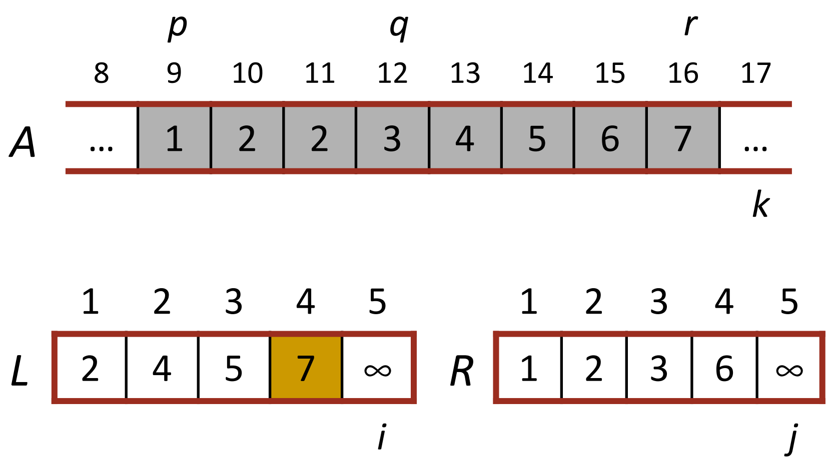


Linear-time merging

```
MERGE (A, p, q, r)
    n1 ← q − p + 1
    n2 ← r − q                                        Θ(1)
    create arrays L[1…n1 + 1] and R[1…n2 + 1]
    for i ← 1 to n1
        do L[i ] ← A[p + i − 1]
    for j ← 1 to n2                                   Θ(n1+n2)
        do R[ j ] ← A[q + j ]
    L[n1 + 1]←∞;  R[n2 + 1]←∞                         Θ(1)
    i ← 1; j ← 1
    for k ← p to r
        do if L[i ] ≤ R[ j ]
            then A[k] ← L[i ]
                i ← i + 1                             Θ(n1+n2)
            else A[k] ← R[ j ]
                j ← j + 1
```


p q r
8 9 10 11 12 13 14 15 16 17
A … 2 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A … 1 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A … 1 2 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j
p q r
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


p q r
8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 1 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
1 2 3 4 5
R 1 2 3 6 ∞
i j
p q r
8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 4 2 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
1 2 3 4 5
R 1 2 3 6 ∞
i j
p q r
8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 4 5 6 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
1 2 3 4 5
R 1 2 3 6 ∞
i j
p q r
8 9 10 11 12 13 14 15 16 17
A ... 1 2 2 3 4 5 3 6 ...
k
1 2 3 4 5
L 2 4 5 7 ∞
1 2 3 4 5
R 1 2 3 6 ∞
i j


p q r
8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

Analyzing divide-and-conquer algorithms

- Use a **recurrence equation** to describe the running time of a divide-and-conquer algorithm.

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

  - $T(n)$ = the running time on a problem of size n.
  - If $n \le c$ for some constant c, the solution takes $\Theta(1)$ time.
  - We divide into a subproblems, each $1/b$ the size of the original.
  - $D(n)$ = the time to divide a size-n problem.
  - $C(n)$ = the time to combine solutions.

Analyzing merge sort$_{1/2}$

- For simplicity, assume that *n* is a power of 2.

$$T(n) = \begin{cases} \theta(1) & \text{if } \; n = 1, \\ 2T(n/2) + \theta(n) & \text{otherwise.} \end{cases}$$

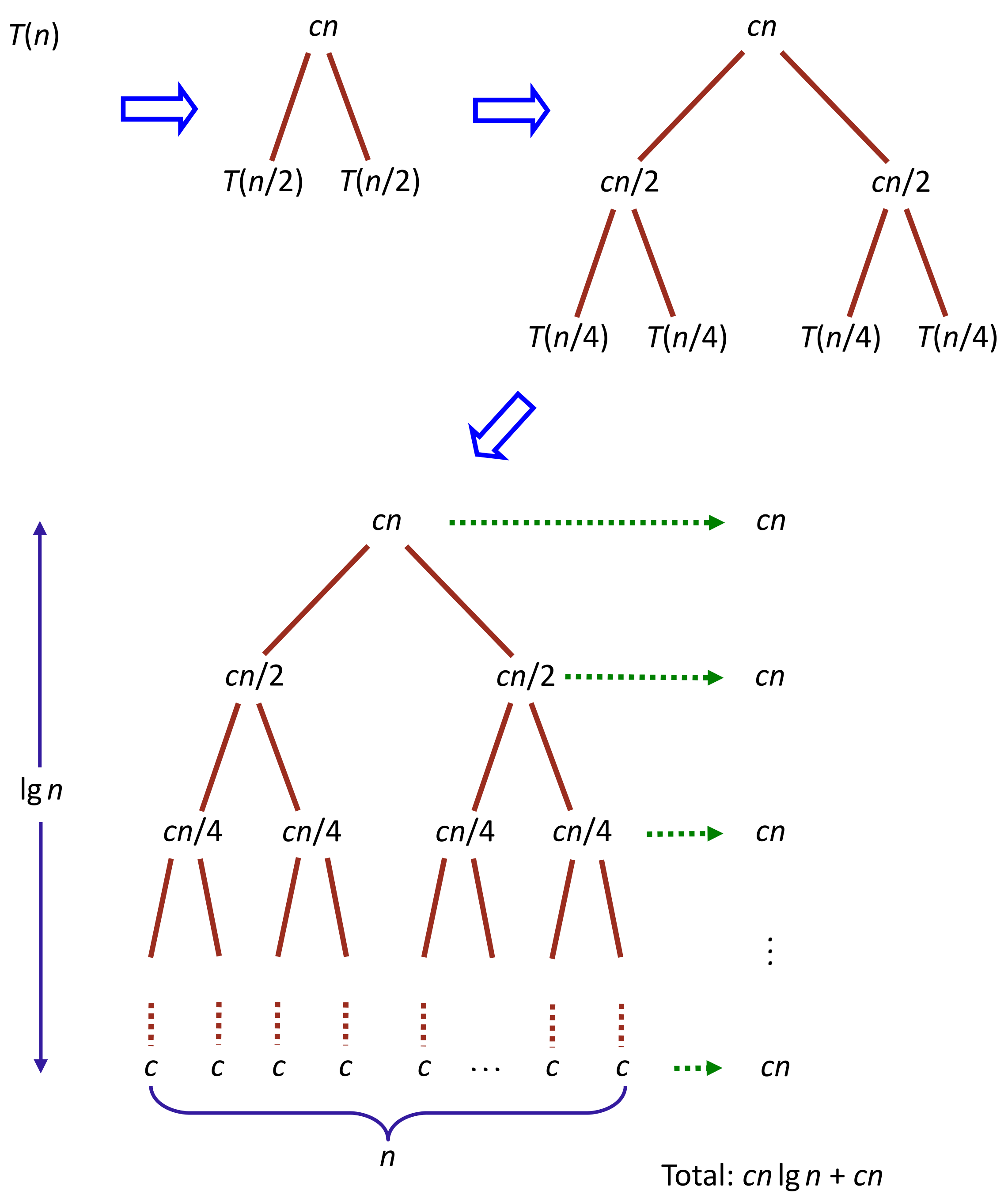
  - The base case occurs when $n = 1$.
  - **Divide**: compute the middle of the subarray, $D(n) = \Theta(1)$.
  - **Conquer**: Recursively solve 2 subproblems, each of size $n/2$
    $\Rightarrow a = 2$ and $b = 2$.
  - **Combine**: MERGE on an *n*-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$.

Analyzing merge sort$_{2/2}$

- Let c be a constant that describes
  - the running time for the base case
  - the time per array element for the divide and combine steps.
- Then, we can rewrite the recurrence as

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

- The next slide shows successive expansions of the recurrence.
  - level i : 2^i nodes, each has a cost of $c(n/2^i)$.
    So, ith level has a cost of $2^i\, c(n/2^i) = cn$.
  - At the bottom level, a tree with h levels has $2^{h-1} = n$ nodes.
    Therefore, $h = \lg n + 1$.
  - The total cost is $cn(\lg n + 1) = cn \lg n + cn = \Theta(n \lg n)$.


$T(n)$
cn
$T(n/2)$ $T(n/2)$
cn
$cn/2$ $cn/2$
$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$
cn
$cn/2$ $cn/2$
$cn/4$ $cn/4$ $cn/4$ $cn/4$
c c c c c $\cdots$ c c
$\lg n$
n
cn
cn
cn
$\vdots$
cn
Total: $cn \lg n + cn$