Ch. 8 Algorithm Efficiency & Sorting

- $O(\ )$: Big-Oh
  - An algorithm is said to take $O(\boldsymbol{f}(n))$
    if
    its running time is upper-bounded by $c\boldsymbol{f}(n)$
  - e.g., $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, …
- Formal definition
  - $O(\boldsymbol{f}(n)) = \{\ \boldsymbol{g}(n) \mid \exists\, c > 0,\ n_0 \geq 0 \text{ s.t. } \forall\, n \geq n_0,\ c\boldsymbol{f}(n) \geq \boldsymbol{g}(n)\ \}$
  - $\boldsymbol{g}(n) \in O(\boldsymbol{f}(n))$이 맞지만 관행적으로 $\boldsymbol{g}(n) = O(\boldsymbol{f}(n))$이라고 쓴다.
- 직관적 의미
  - $\boldsymbol{g}(n) = O(\boldsymbol{f}(n)) \Rightarrow \boldsymbol{g}$ grows no faster than $\boldsymbol{f}$

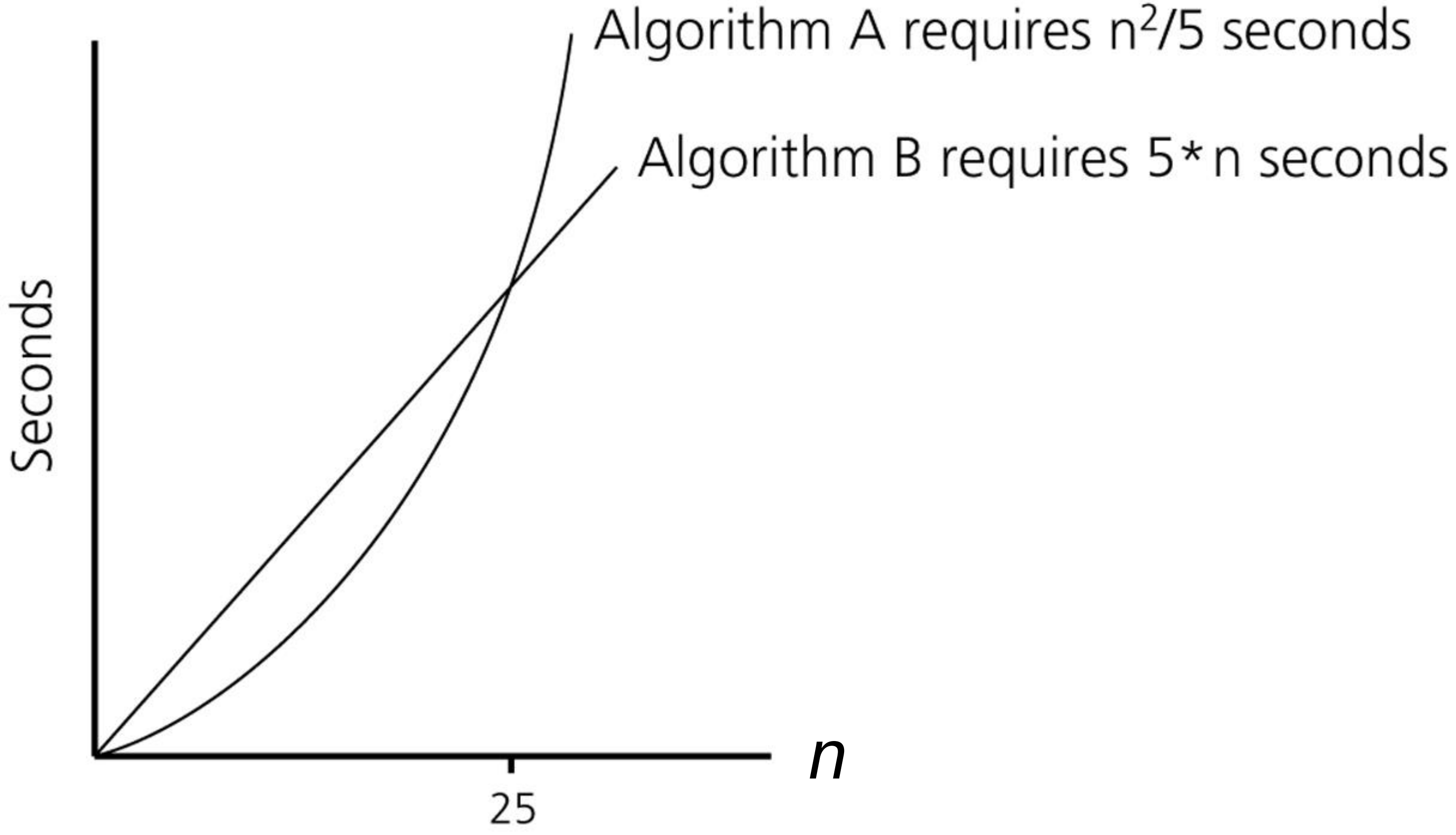
$\Omega(f(n))$

- 적어도 $f(n)$의 비율로 증가하는 함수
- $O(f(n))$과 대칭적

$\Theta(f(n))$

- $f(n)$의 비율로 증가하는 함수
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

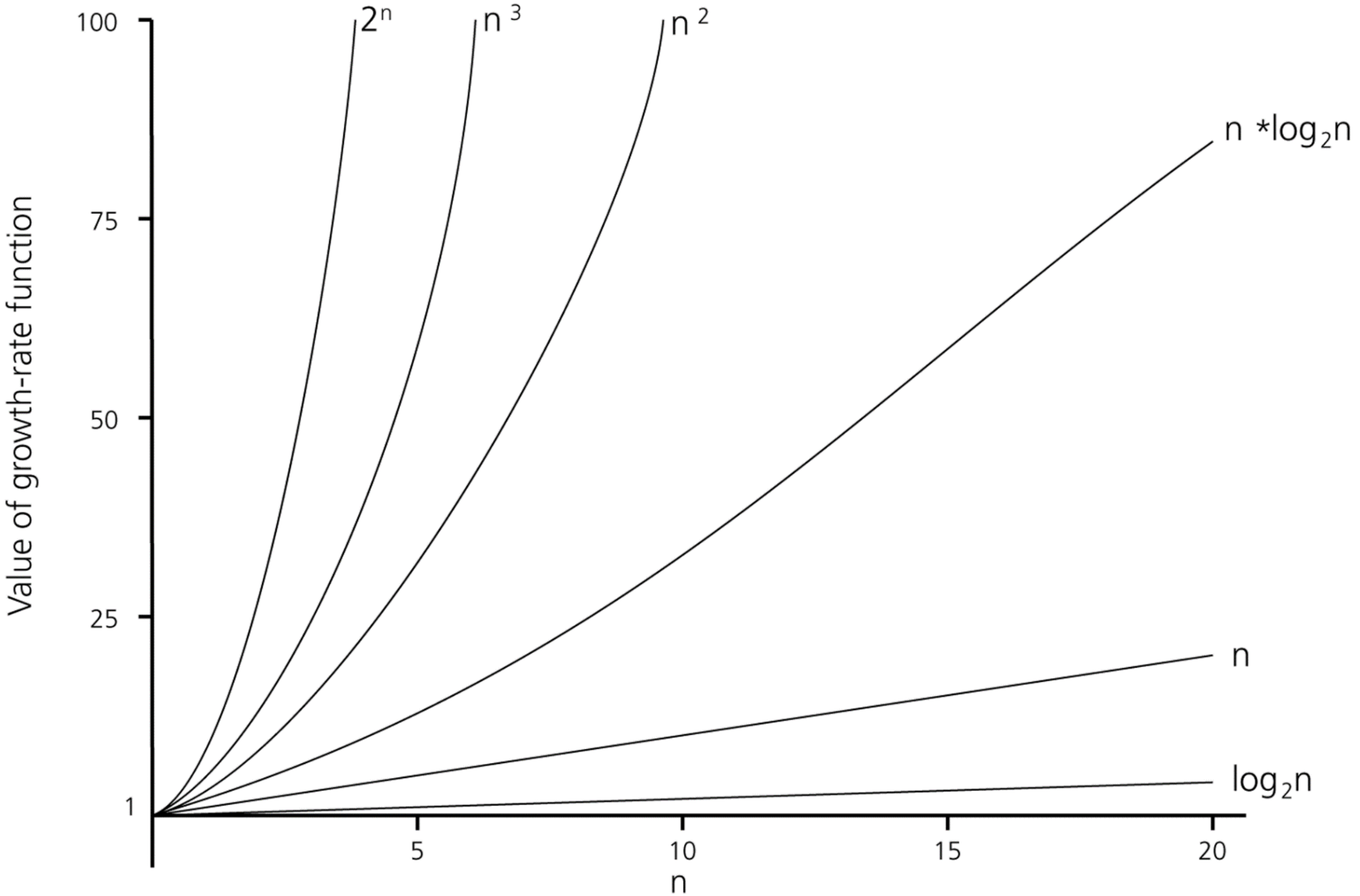
Time requirements as a function of the problem size *n*



A comparison of growth-rate functions

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A comparison of growth-rate functions



Types of Running-Time Analysis

- Worst-case analysis
  - Analysis for the worst-case input(s)
- Average-case analysis
  - Analysis for all inputs
  - More difficult to analyze
- Best-case analysis
  - Analysis for the best-case input(s)
  - Mostly not meaningful

Running Time for Search in an Array

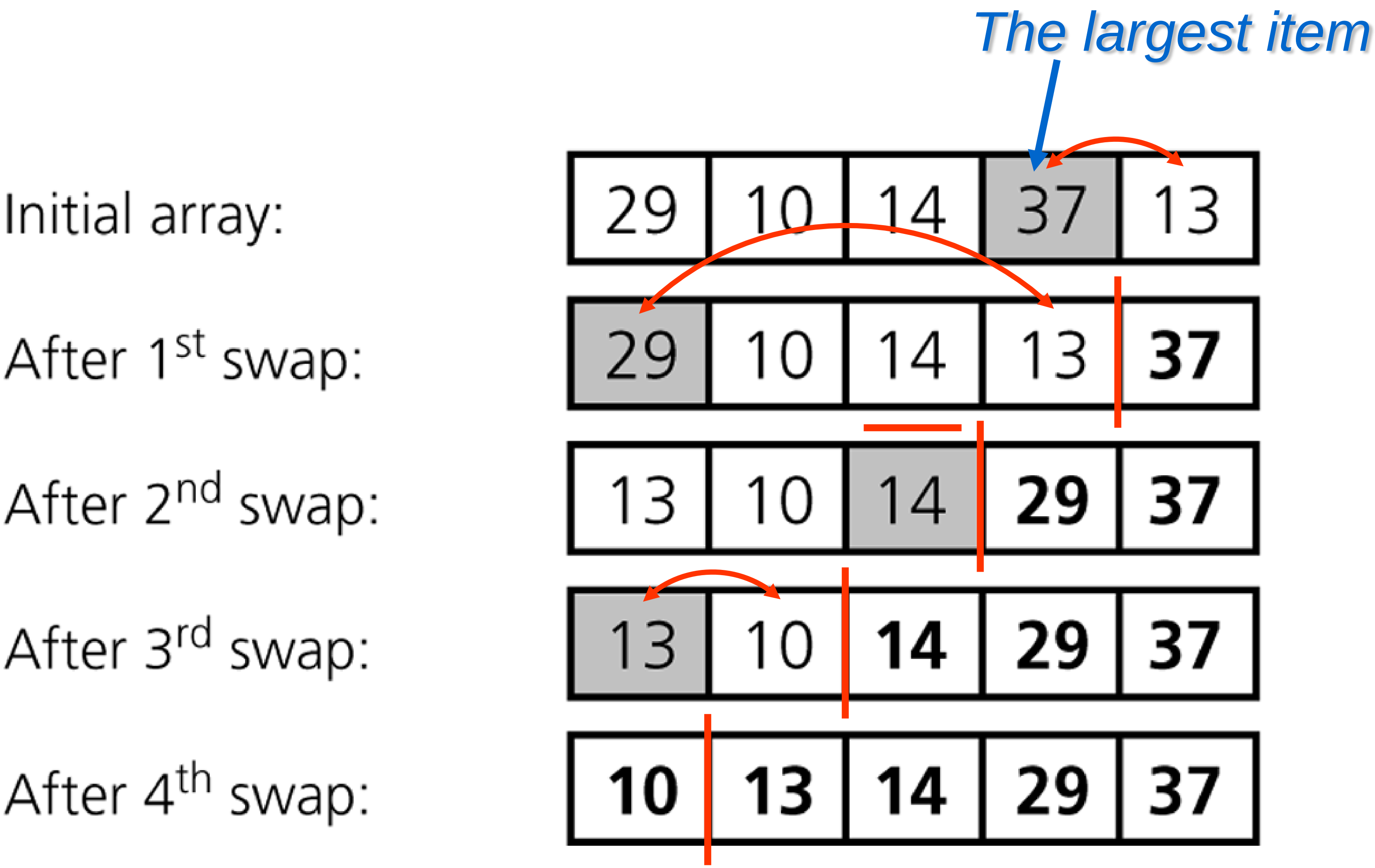
- Sequential search
  - Worst case: $\Theta(n)$
  - Average case: $\Theta(n)$
- Binary search
  - Worst case: $\Theta(\log n)$
  - Average case: $\Theta(\log n)$ ← 각자 확인 요망!

Sorting Algorithms

- 대부분 $\Theta(n^2)$과 $\Theta(n\log n)$ 사이
- Input이 특수한 성질을 만족하는 경우에는 $\Theta(n)$ sorting도 가능
  - E.g., input이 $-O(n)$과 $O(n)$ 사이의 정수

Selection Sort

- An iteration
  - Find the largest item
  - Swap it to the rightmost place
  - Exclude the rightmost item
- Repeat the above iteration until only one item remains



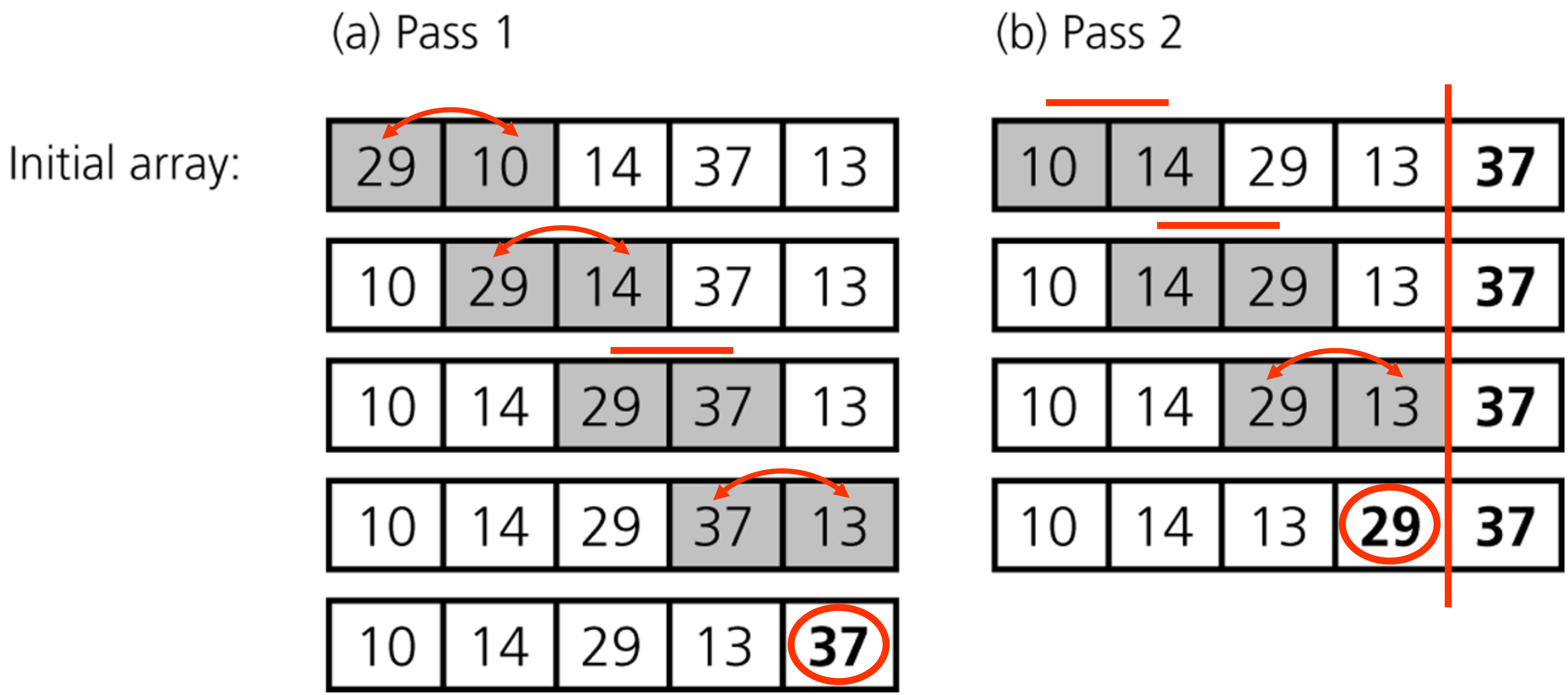
✓ Running time: $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = \Theta(n^2)$ ← Worst case, Average case

```
selectionSort(theArray[ ], n)
{
        for (last = n-1; last >=1; last--) {
                largest = indexOfLargest(theArray, last+1);
                Swap theArray[largest] & theArray[last];
        }
}
indexOfLargest(theArray, size)
{
        largest = 0;
        for (i = 1; i < size; ++i) {
                if (theArray[i] > theArray[largest]) largest = i;
        }
}
```

✓ Running time: 두 함수의 **for** loop의 iteration 수의 합이 좌우
— indexOfLargest가 *n*-1 번 call 되고,
call 될 때마다 indexOfLargest의 수행시간은
한 단계씩 가벼워진다.

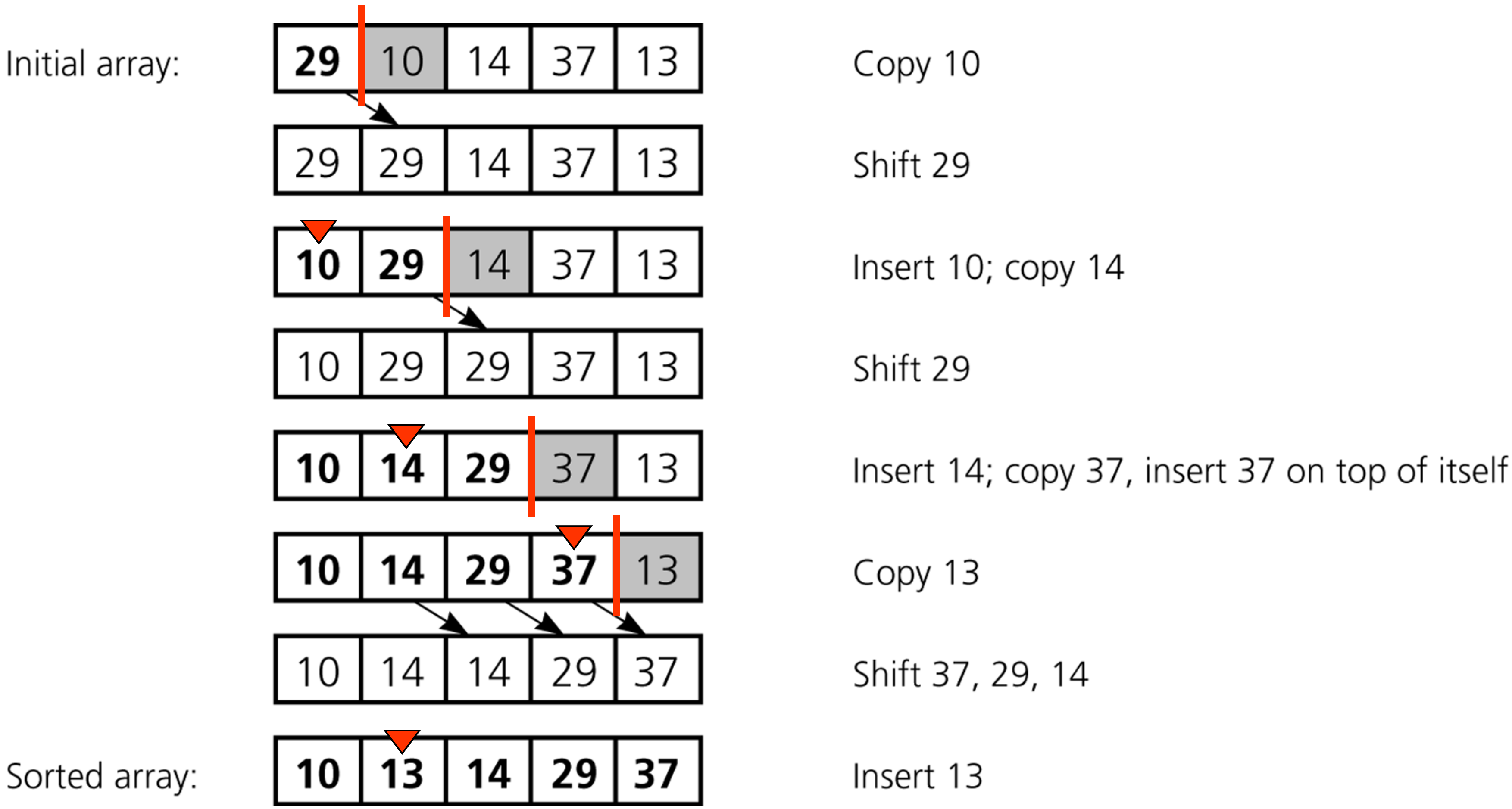
✓ $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = \Theta(n^2)$

Bubble Sort



✓ Running time: $(n\text{-}1)+(n\text{-}2)+\cdots+2+1 = \Theta(n^2)$ ← Worst case, Average case

Insertion Sort

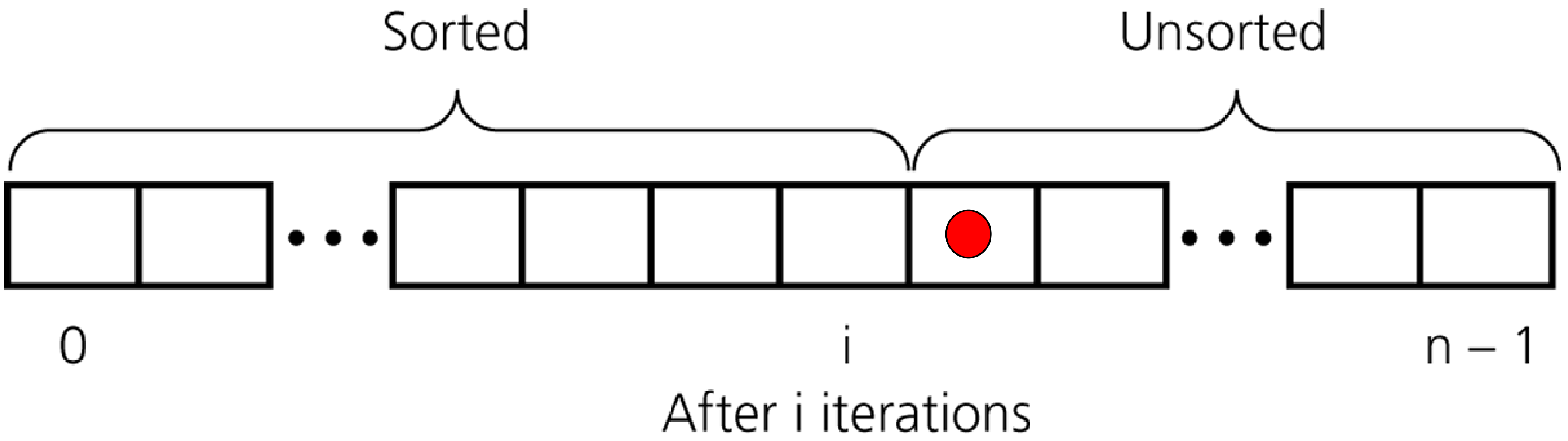


✓ Running time: $\Theta(n^2)$

Worst case: $1+2+\cdots+(n-2)+(n-1)$

Average case: $\frac{1}{2}\,(1+2+\cdots+(n-2)+(n-1))$

Insertion sort에서 중간의 한 시점

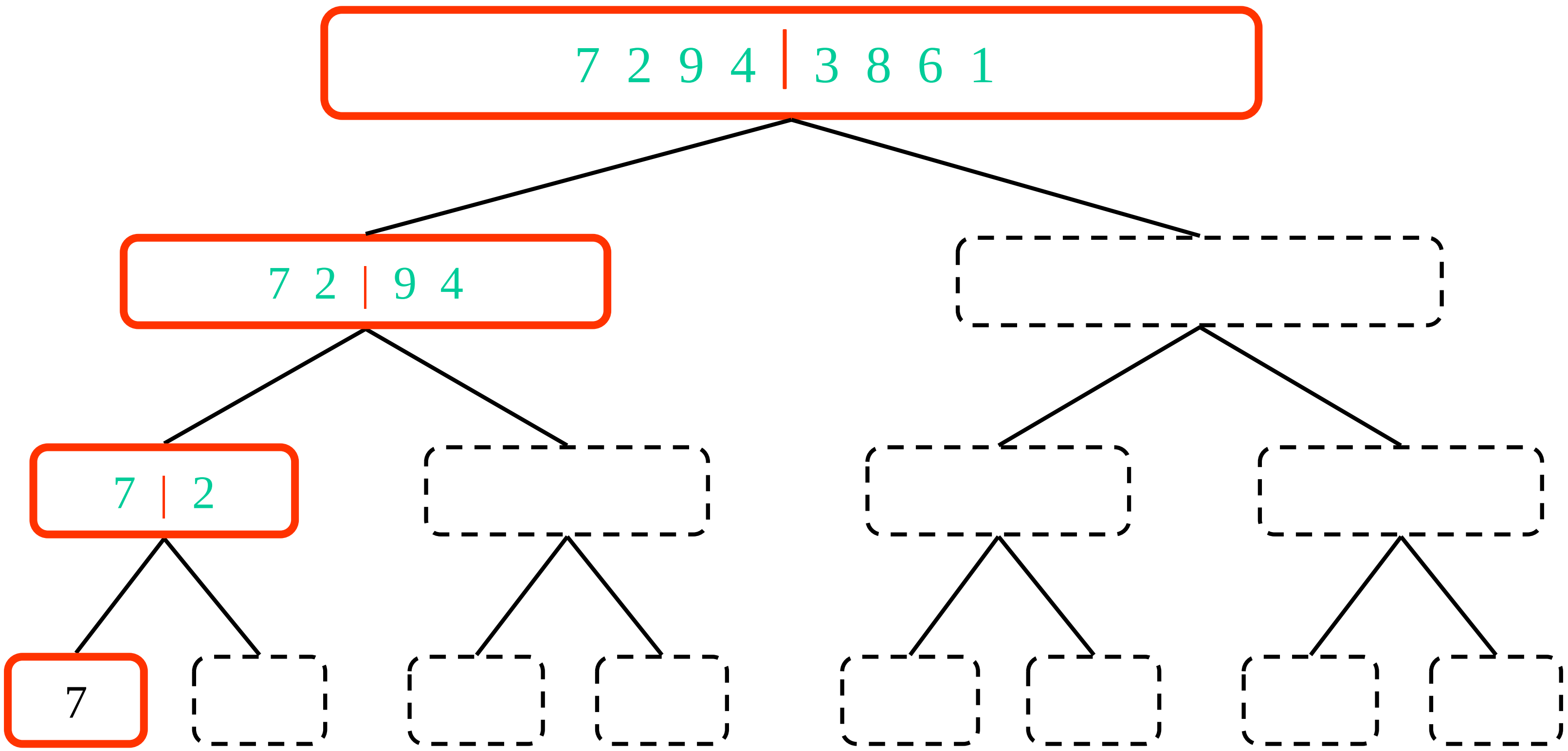


Mergesort

```
mergeSort(S)
{       // Input: sequence S with n elements
        // Output: sorted sequence S
        if (S.size( ) > 1) {
            Let S1, S2 be the 1st half and 2nd half of S, respectively;
            mergeSort(S1);
            mergeSort(S2);
            S ← merge(S1, S2);
        }
}

merge(S1, S2)
{       sorting된 두 sequence S1, S2 를 합쳐
        sorting 된 하나의 sequence를 만든다
}
```

Animation (Mergesort)

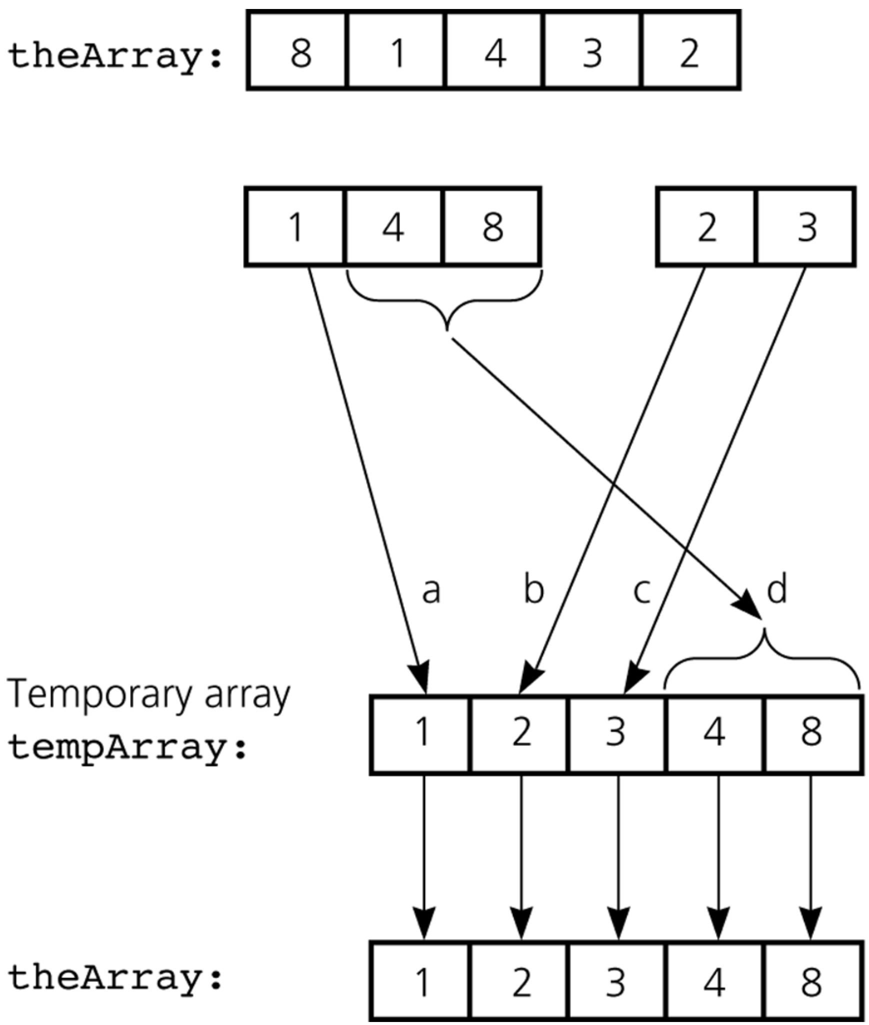


Animation (Mergesort)

1 2 3 4 6 7 8 9

✓ Running time: $\Theta(n\log n)$

Merge는 보조 array가 필요하다



Divide the array in half

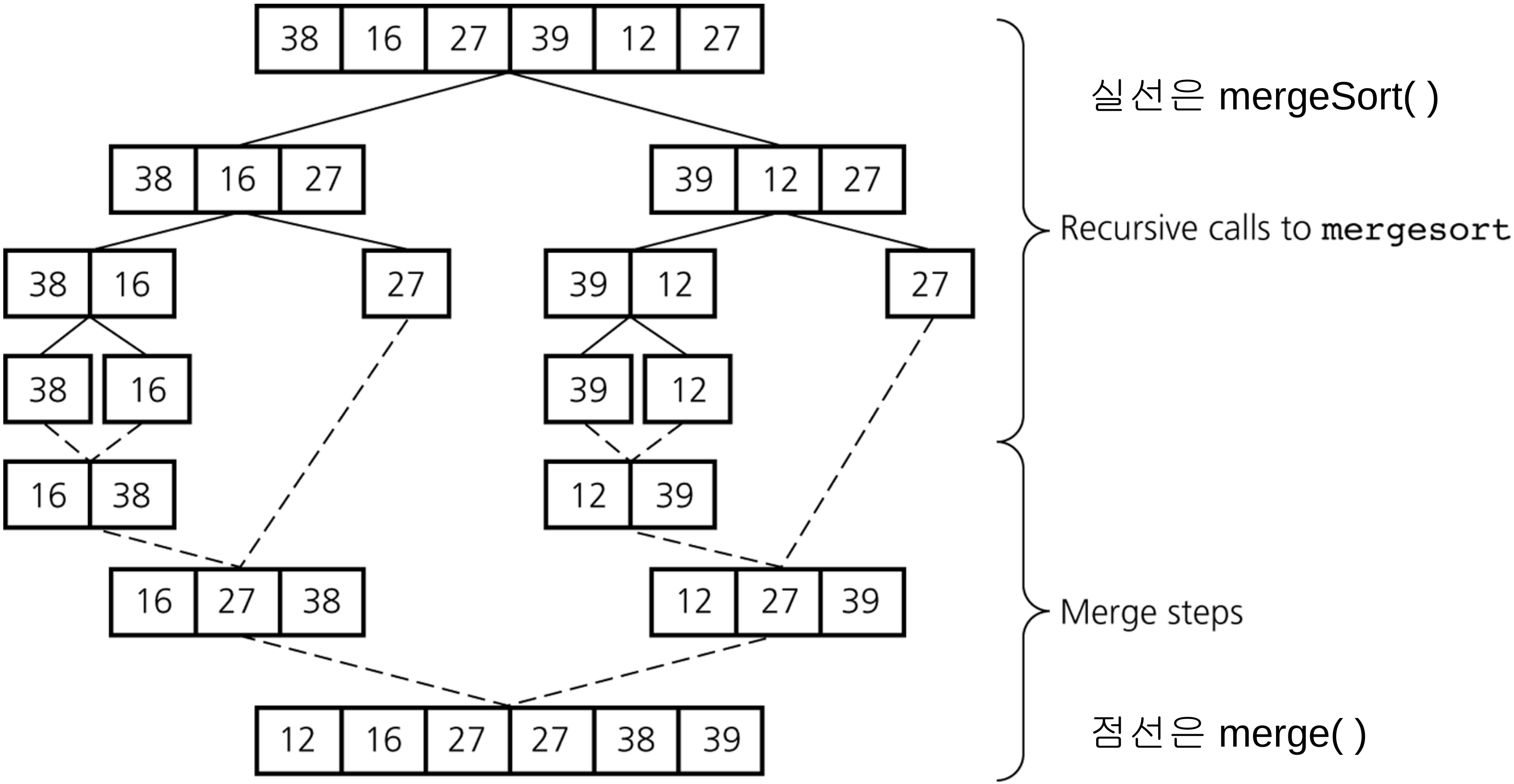
Sort the halves

Merge the halves:

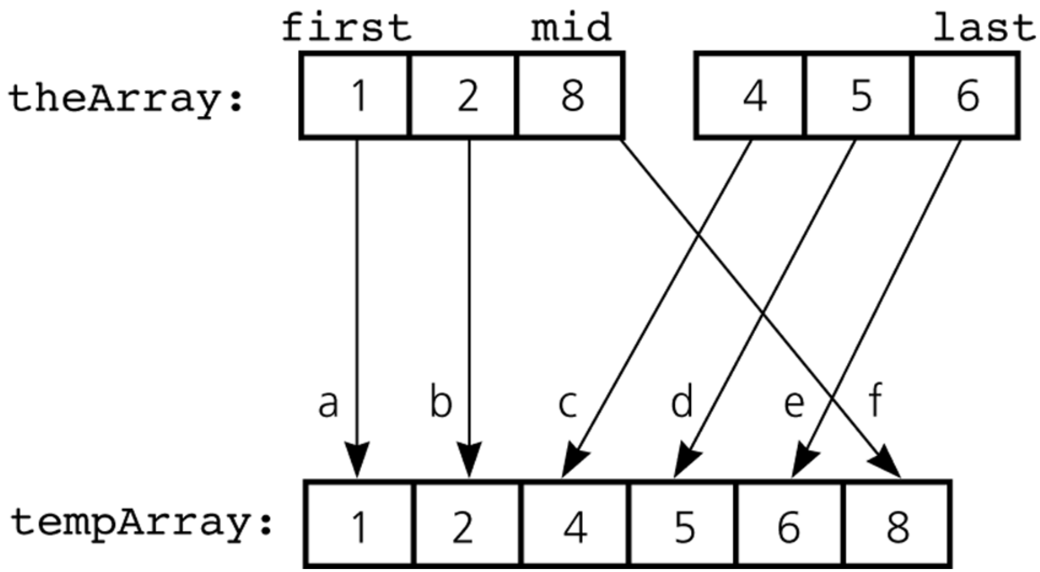
a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

A mergesort of an array of six integers



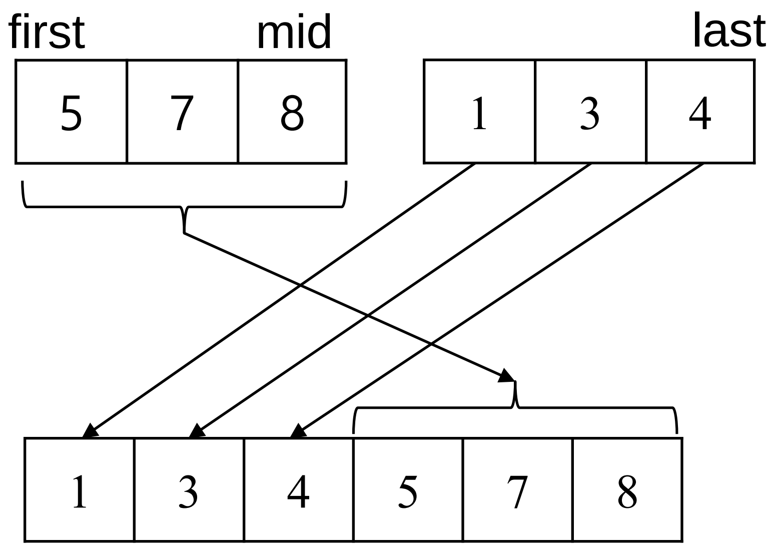
A worst-case instance of the merge step in `mergesort`



Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

A best-case instance of the merge step in ***`mergesort`***



Quicksort

```
quickSort(S)
{       // Input: sequence S with n elements
        // Output: sorted sequence S
        if (S.size( ) > 1) {
            p ← pivot of S;
            (L, R) ← partition(S, p); // L: left partition, R: right partition
            quickSort(L);
            quickSort(R);
            return L • p • R; // concatenation
        }
}
partition(S, p)
{       sequence S에서 p보다 작은 item은 partition L로,
        p보다 크거나 같은 item은 partition R로 분리.
}
```

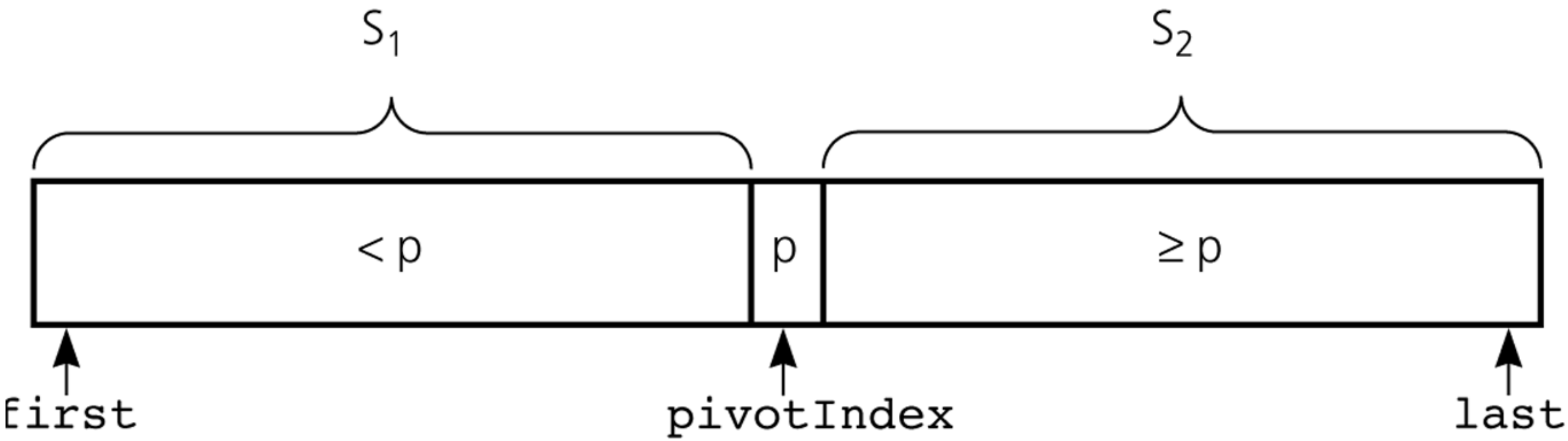
Animation (Quicksort)

1 2 3 4 5 6 8 9

✓ Average-case running time: $\Theta(n \log n)$

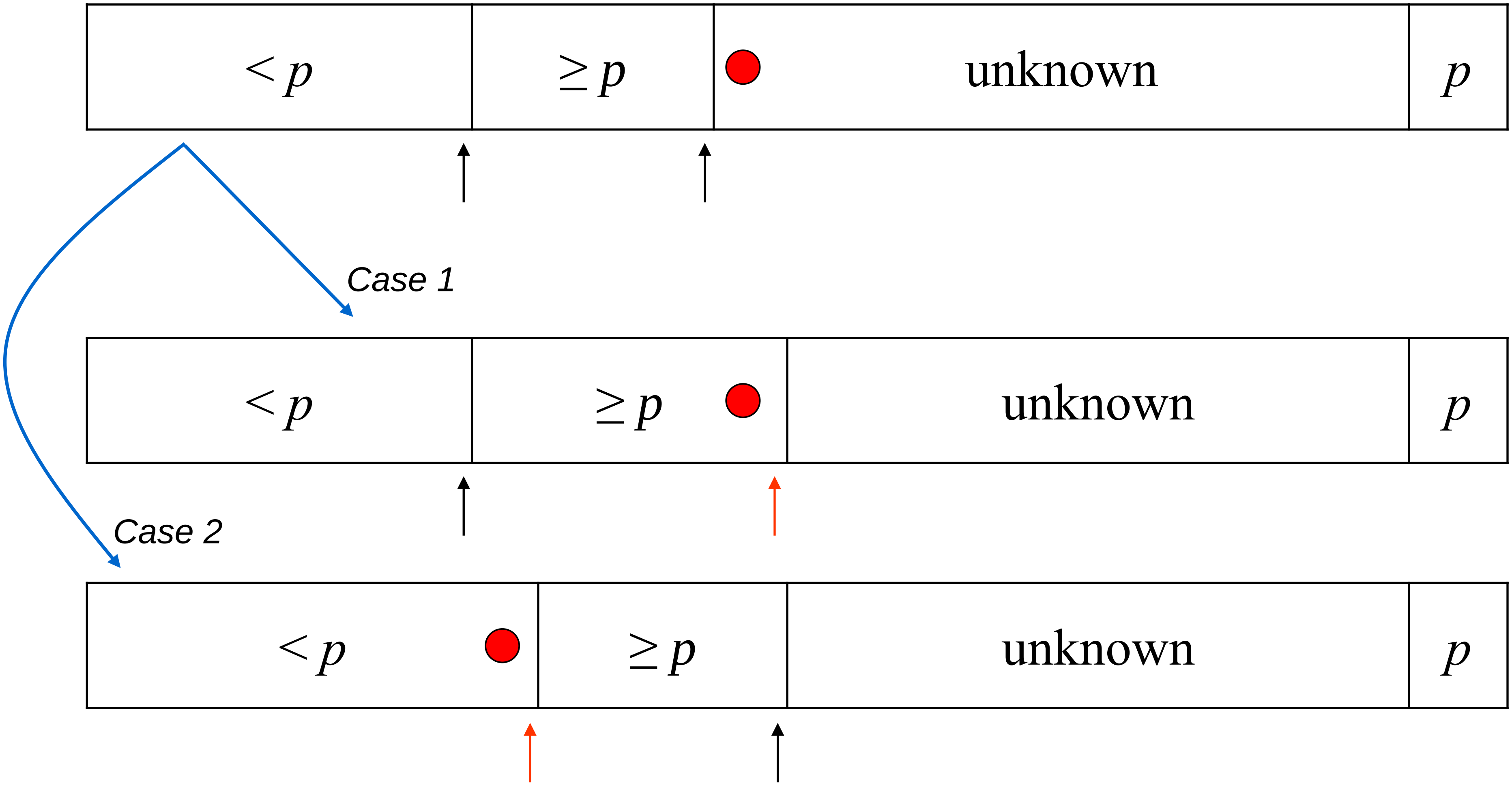
✓ Worst-case running time: $\Theta(n^2)$

A partition with a pivot



✓ Partitioning 방법은 다양하다

Partition의 예. 중간의 한 시점.

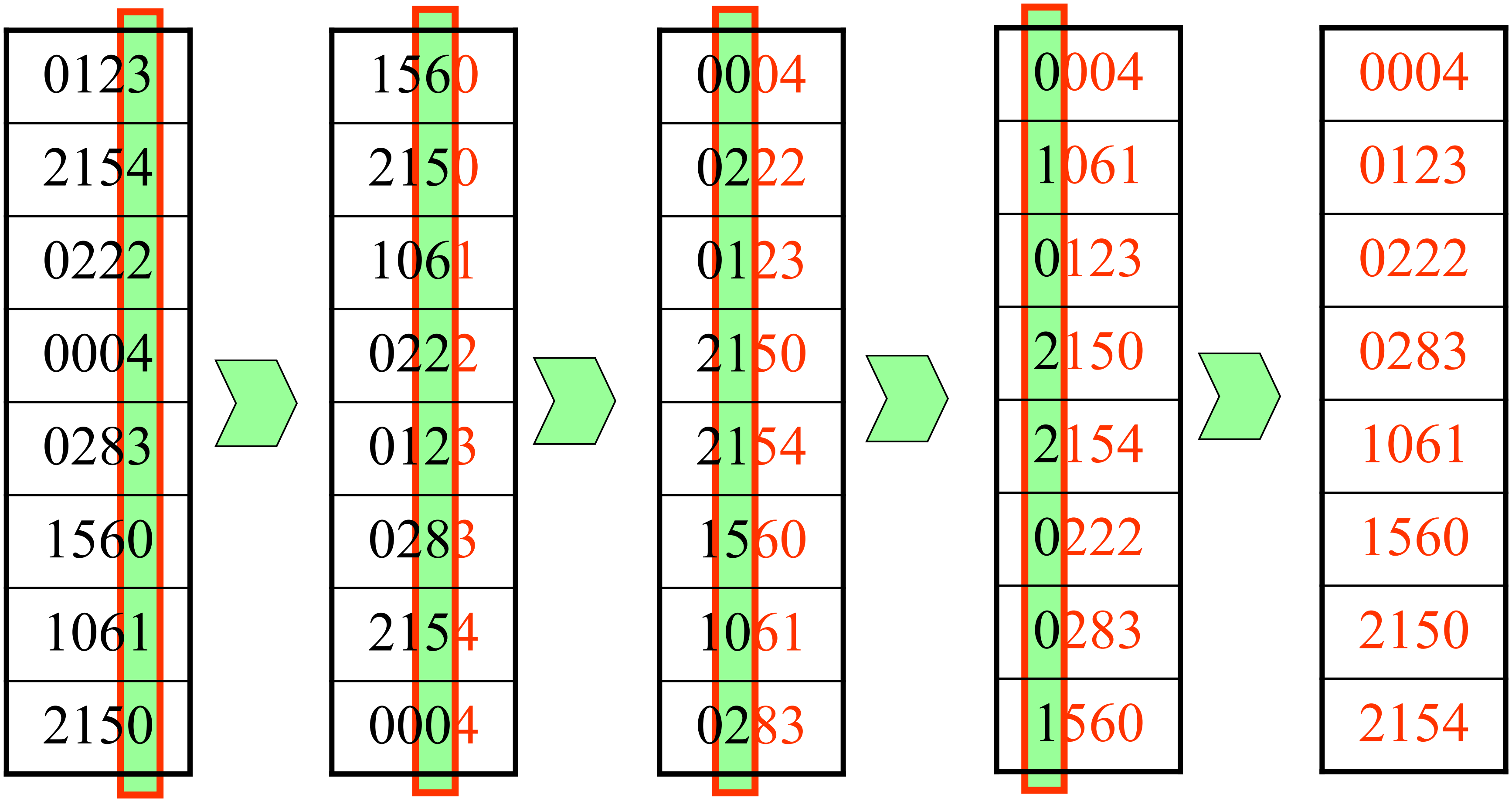


Radix Sort

```
radixSort(A[ ], d)
{         // Sort n d-digit integers in the array A[ ]
          for (j = d downto 1) {
                    Do a stable sort on A[ ] by jth digit;
          }
}
```

✓ Stable sort

— 같은 값을 가진 item들은 sorting 후에도 원래의 순서가 유지되는 성질을 가진 sorting



✓ Running time: $\Theta(n)$ ← d: a constant

Comparison of Sorting Efficiency in $\theta(\,)$

	Worst Case	Average Case
Selection Sort	n^2	n^2
Bubble Sort	n^2	n^2
Insertion Sort	n^2	n^2
Mergesort	$n\log n$	$n\log n$
Quicksort	n^2	$n\log n$
Radix Sort	n	n
Heapsort	$n\log n$	$n\log n$