Lecture #12 | 정렬 알고리즘 (sorting algorithm)

SE213 프로그래밍 (2019)

공지

- 4th assignment
  - 5/23 ~ 6/3 (midnight)
  - 3 problems at dgist.elice.io

지난 시간에 다룬 내용

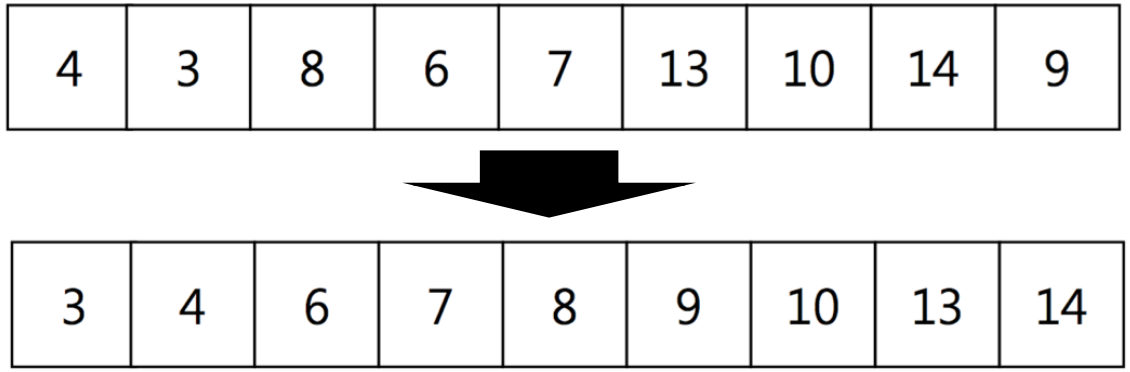
- 알고리즘이란
- 검색(탐색) 알고리즘
  - 순차 검색 (sequential search)
  - 이진 검색 (binary search)
- 알고리즘 복잡도

이번 시간에 다룰 내용

- 정렬 알고리즘 (sorting algorithm)
  - 선택 정렬 (selection sort)
  - 병합 정렬 (merge sort)
  - 퀵 정렬 (quick sort)
- 정렬 알고리즘 비교

정렬 (sorting): 문제의 정의

- 아이템들을 일정한 순서에 따라 정렬하는 것
  - 주로 리스트에 저장되어 있는 아이템을 정렬
  - 정렬된 아이템들을 원래 리스트 혹은 별도의 리스트에 저장
  - 정렬을 할 기준이 제공되어야 함 (예: 숫자 크기 순, 알파벳 순)



- 정렬 알고리즘의 고려 사항: 시간 복잡도 & 공간 복잡도

python의 정렬 함수

- `sorted(iterable, key=None, reverse=False)`
  - python의 내장함수 (built-in function)
  - 매개변수
    - `iterable`: `list`, `tuple`, `str` 등의 자료형이 올 수 있음 → **원소는 그대로 유지**
    - `key`: 정렬의 규칙을 임의로 바꿀 수 있음
    - `reverse`: `True`인 경우 역순으로 정렬
  - 반환값: 인자로 전달된 원소들을 순서대로 포함한 **새로운** 리스트
- `list.sort(key=None, reverse=False)`
  - 리스트의 메소드로, **리스트 내부에서 원소를 정렬**
  - 매개변수: `key`, `reverse`는 `sorted()`와 동일
  - 반환값: 없음 (None)

예시: sort() and sorted()

```
t = [4, 3, 8, 6, 7, 13, 10, 14, 9]
t2 = sorted(t)
print(t)
print(t2)
t.sort()
print(t)
```

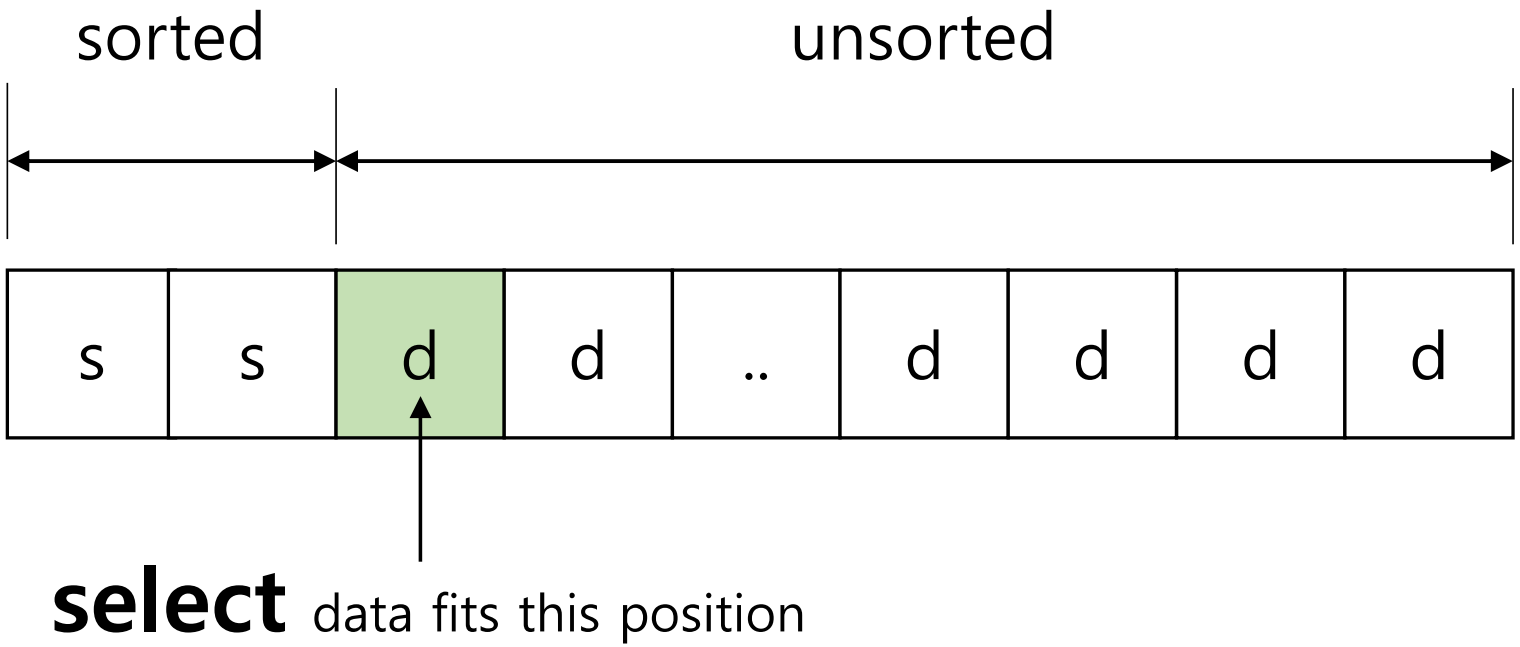
```
[4, 3, 8, 6, 7, 13, 10, 14, 9]
[3, 4, 6, 7, 8, 9, 10, 13, 14]
[3, 4, 6, 7, 8, 9, 10, 13, 14]
```

정렬 알고리즘의 종류

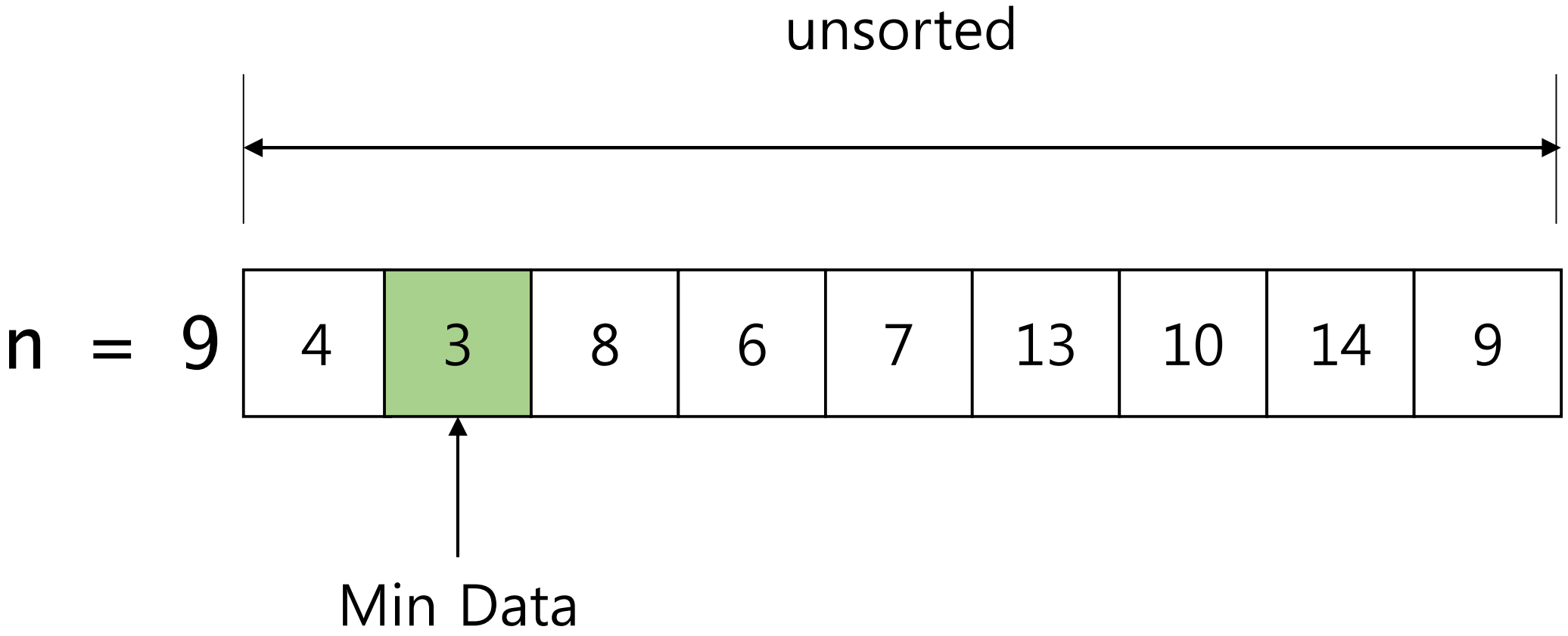
- Insertion sort
- **Selection sort**
- **Merge sort**
- **Quick sort**
- Heap sort
- Bubble sort
- Shell sort
- Bucket sort
- Radix sort
- …
- 참고: https://en.wikipedia.org/wiki/Sorting_algorithm

선택 정렬

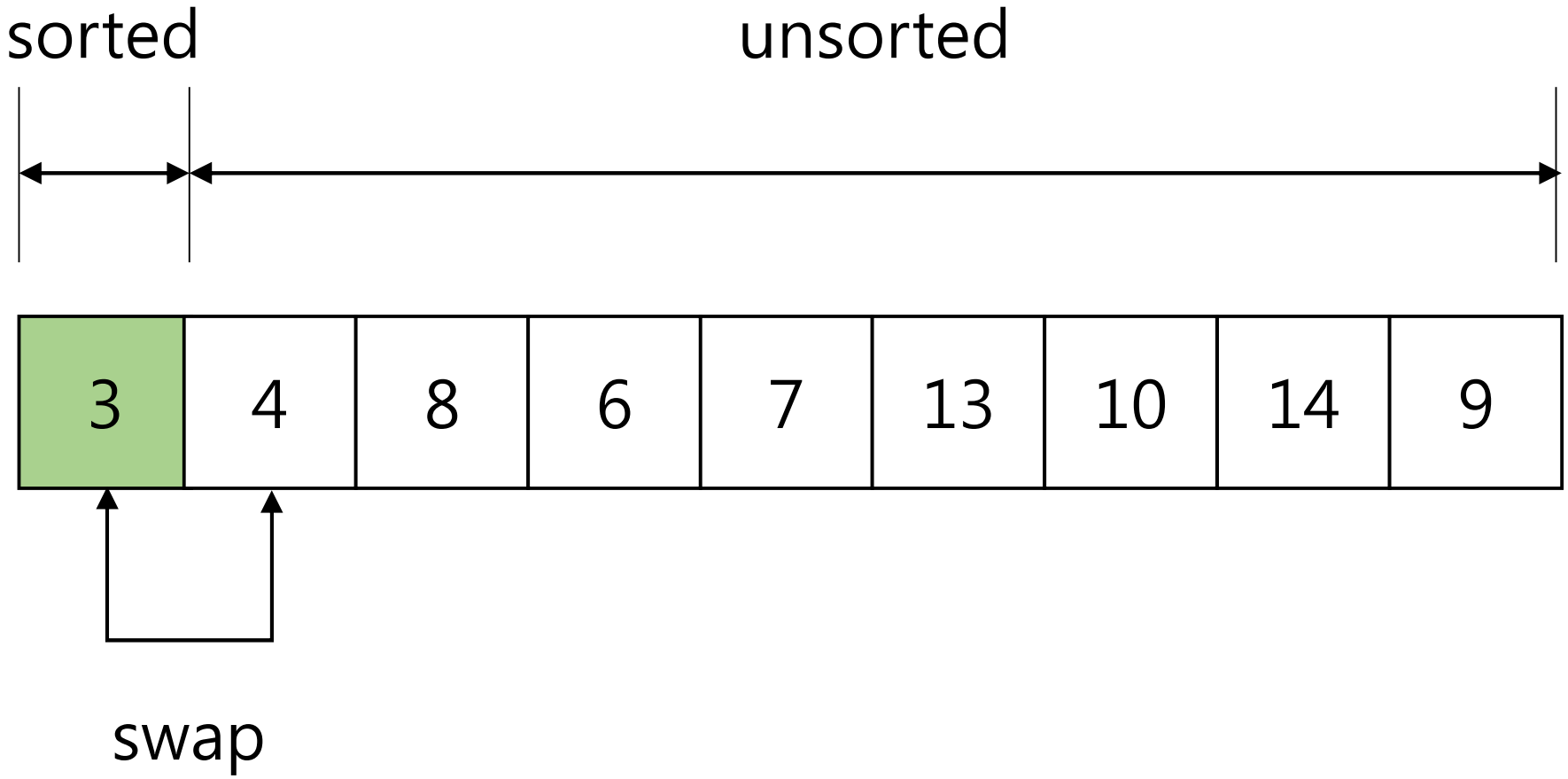
- 모든 아이템들이 정렬이 될 때까지 다음 작업을 반복
  - 정렬이 되지 않은 구간에서 가장 작은 수를 찾음
  - 정렬이 되지 않은 구간의 첫번째 숫자와 가장 작은 수를 교환



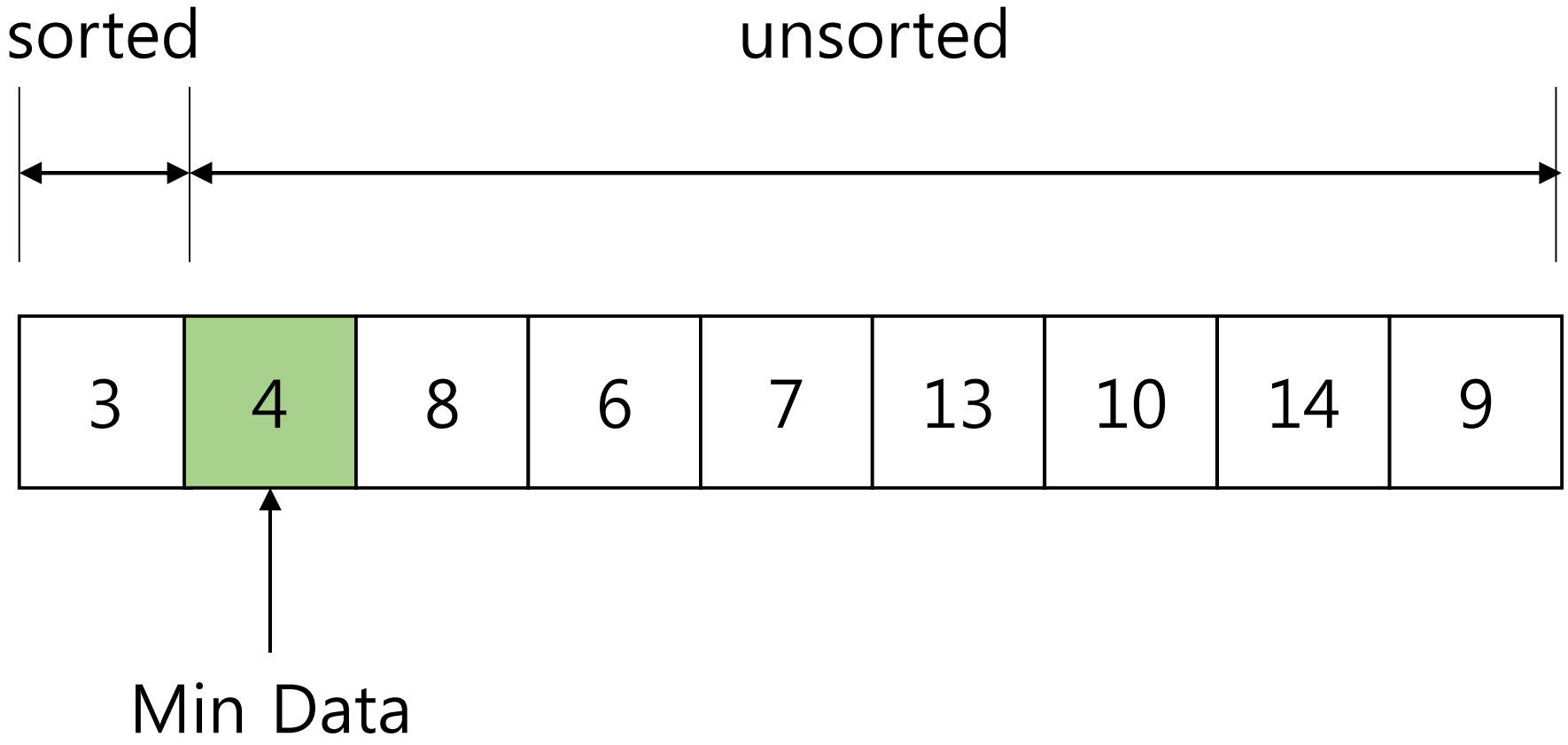
예시: 선택 정렬 (1/17)



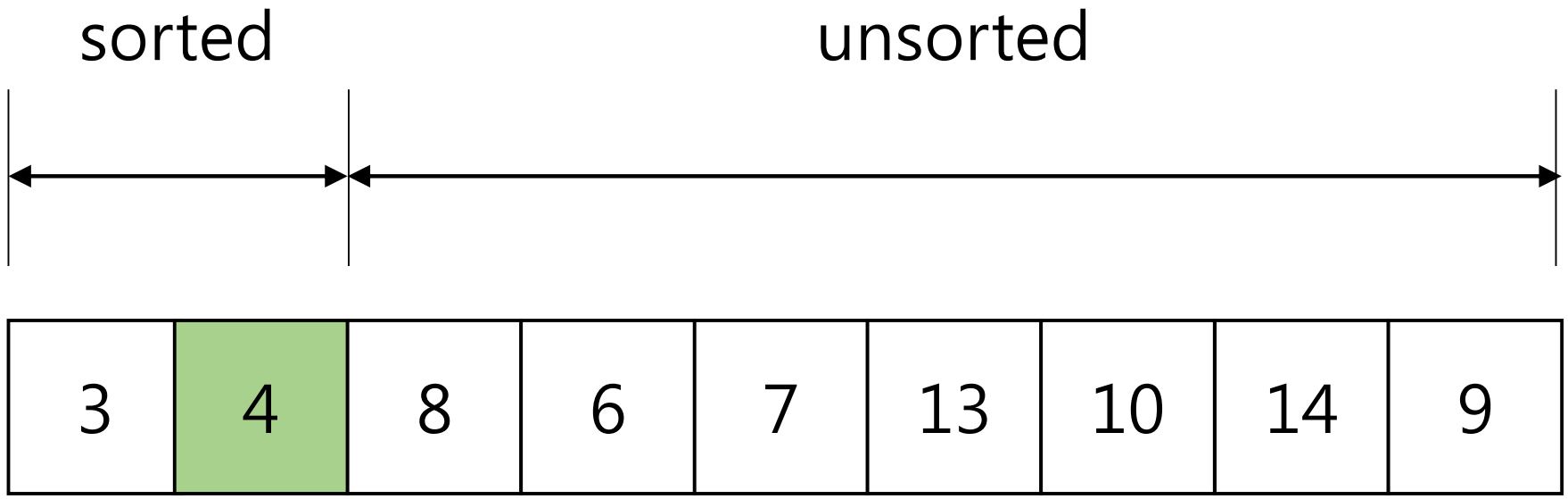
예시: 선택 정렬 (2/17)



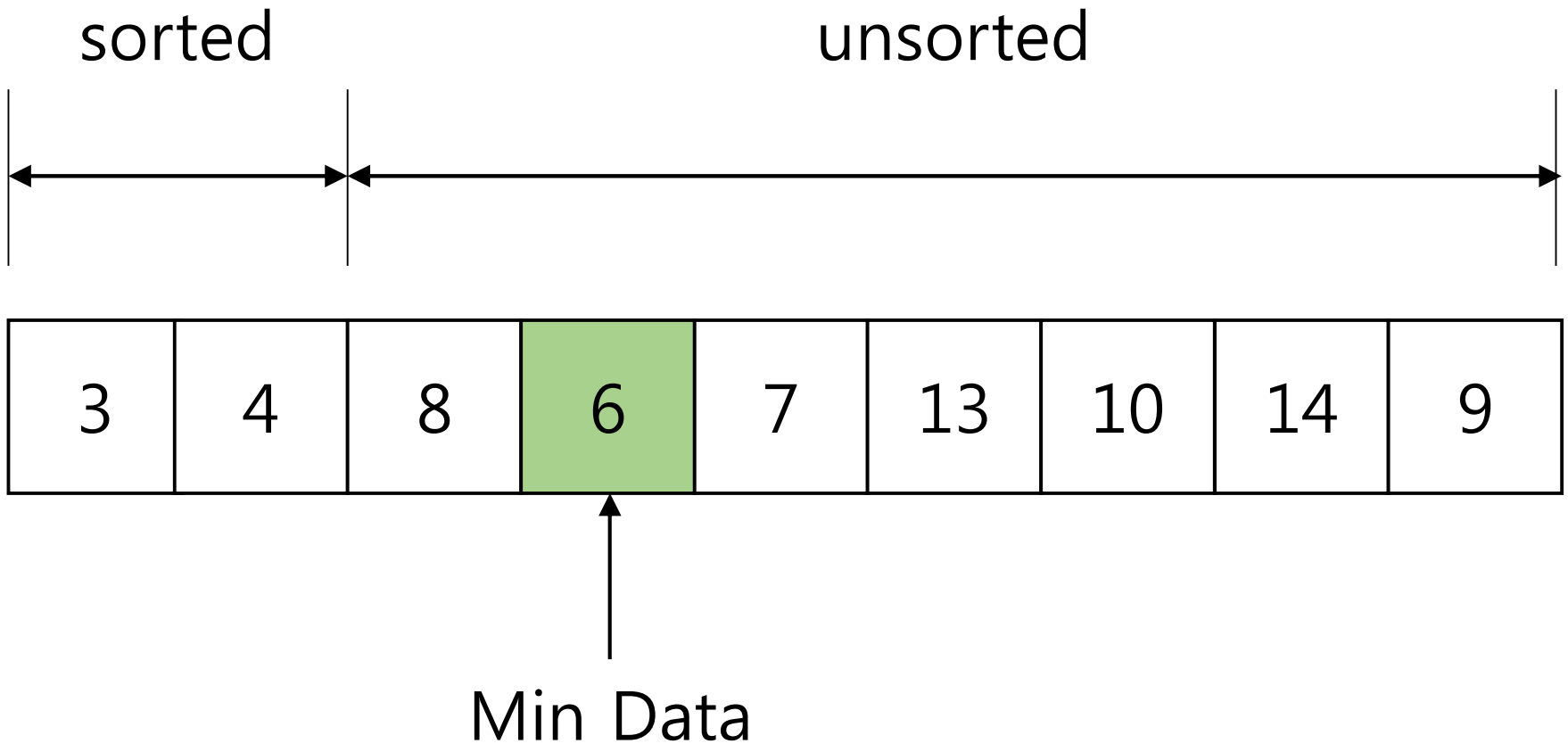
예시: 선택 정렬 (3/17)



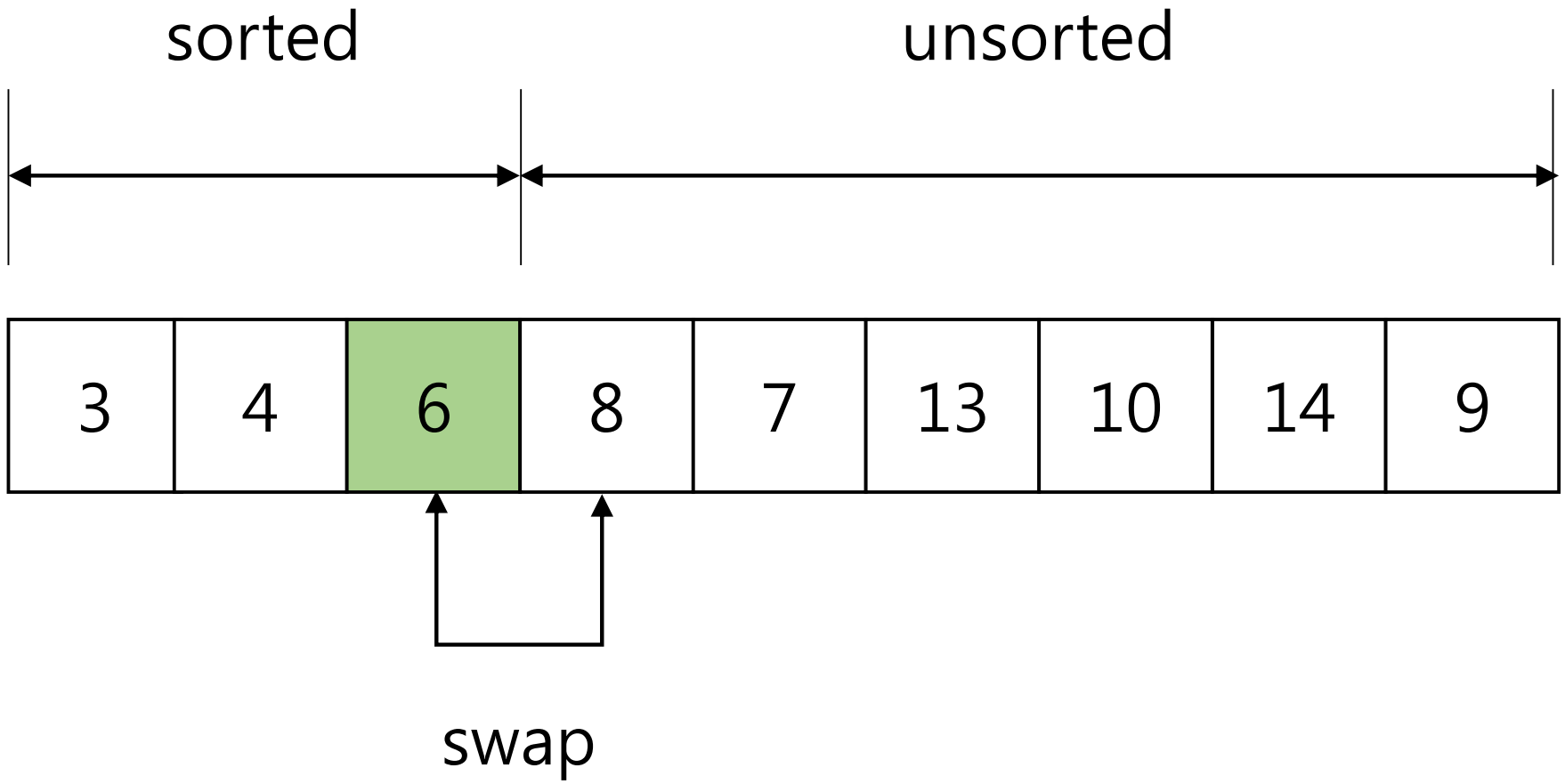
예시: 선택 정렬 (4/17)



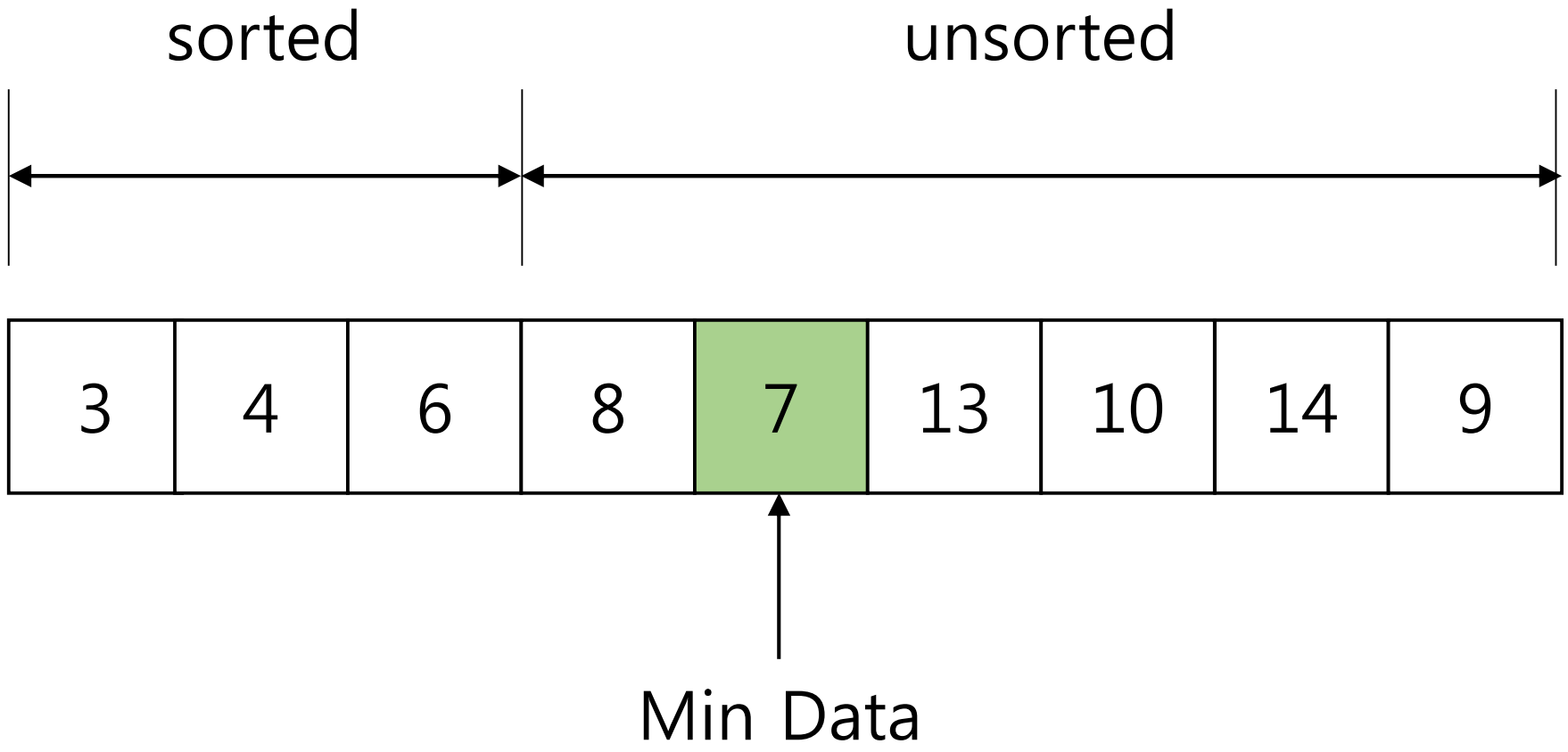
예시: 선택 정렬 (5/17)



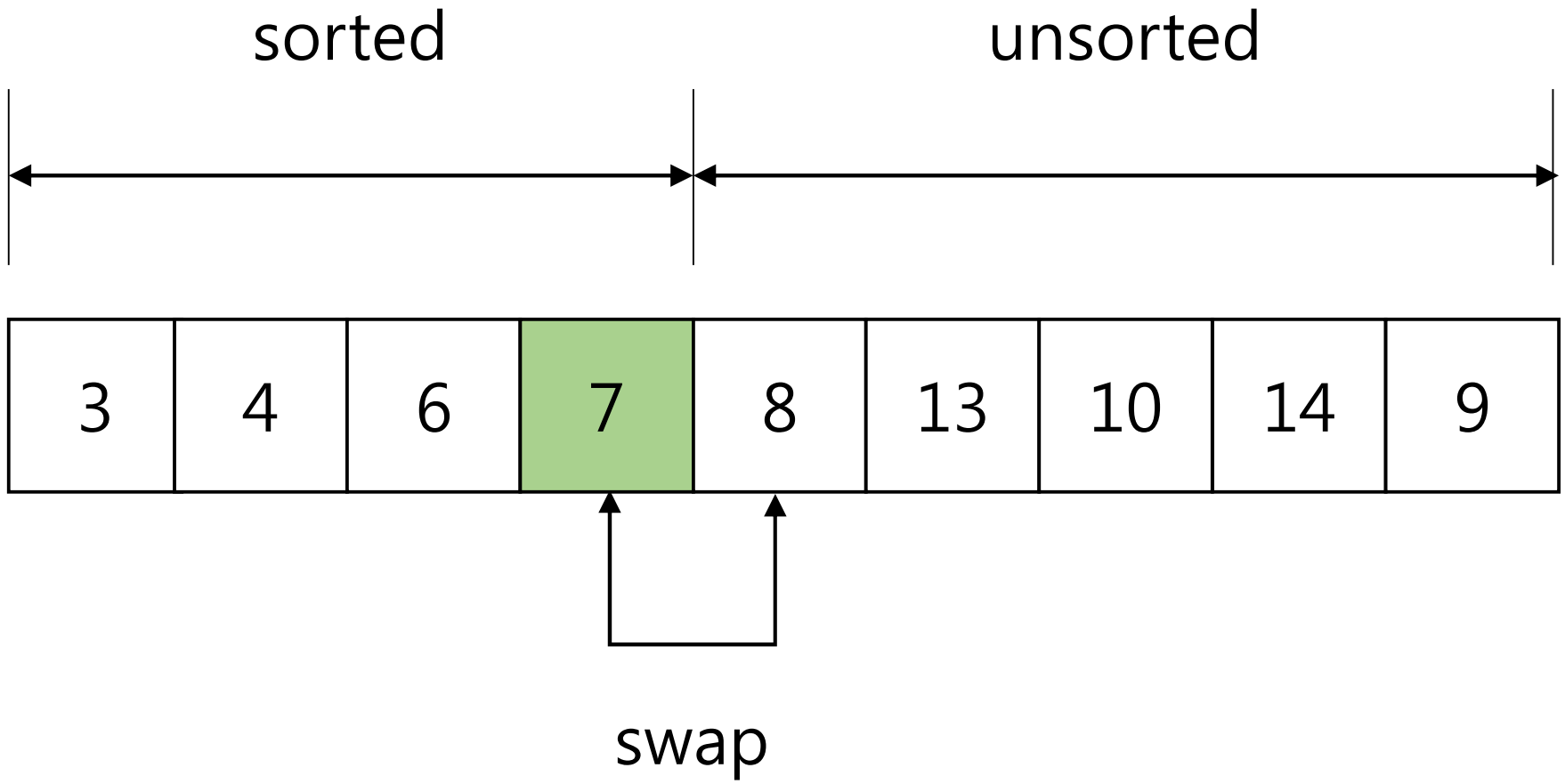
예시: 선택 정렬 (6/17)



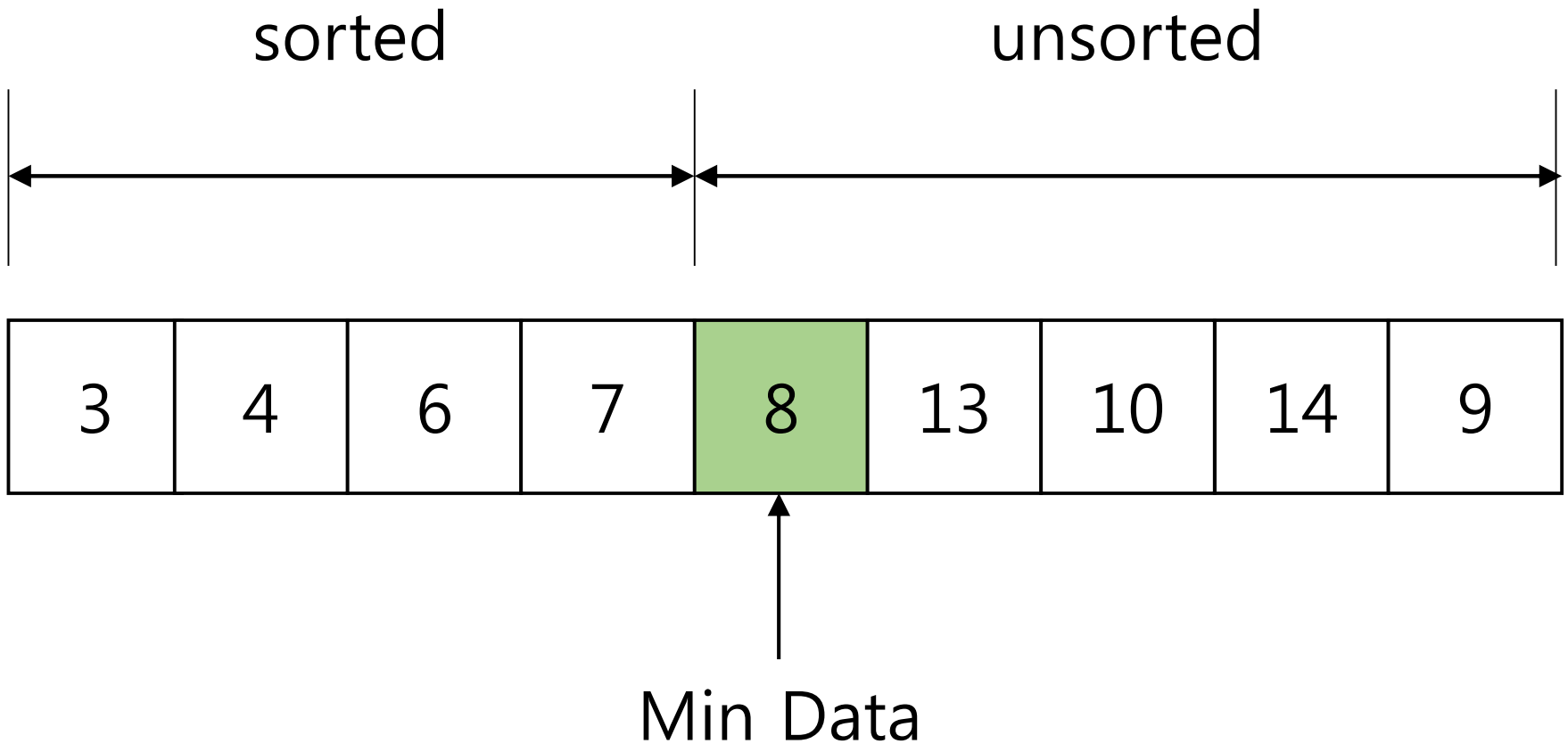
예시: 선택 정렬 (7/17)



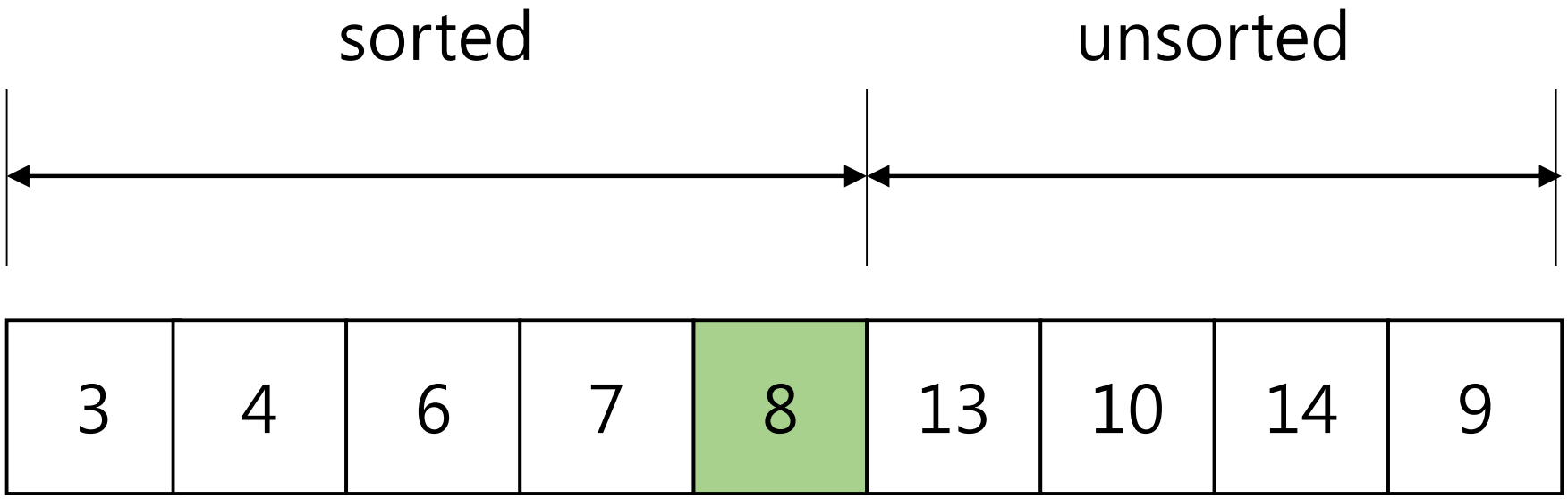
예시: 선택 정렬 (8/17)



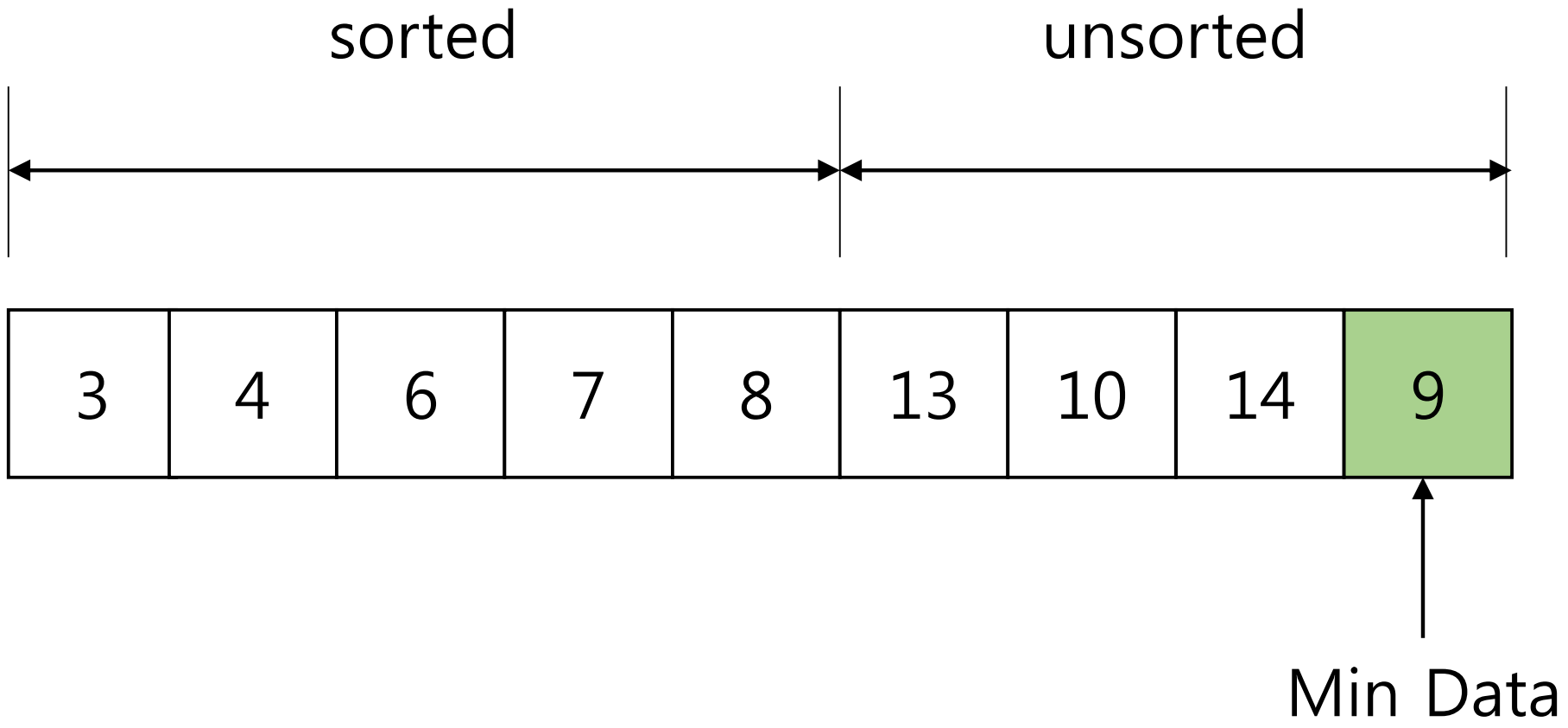
예시: 선택 정렬 (9/17)



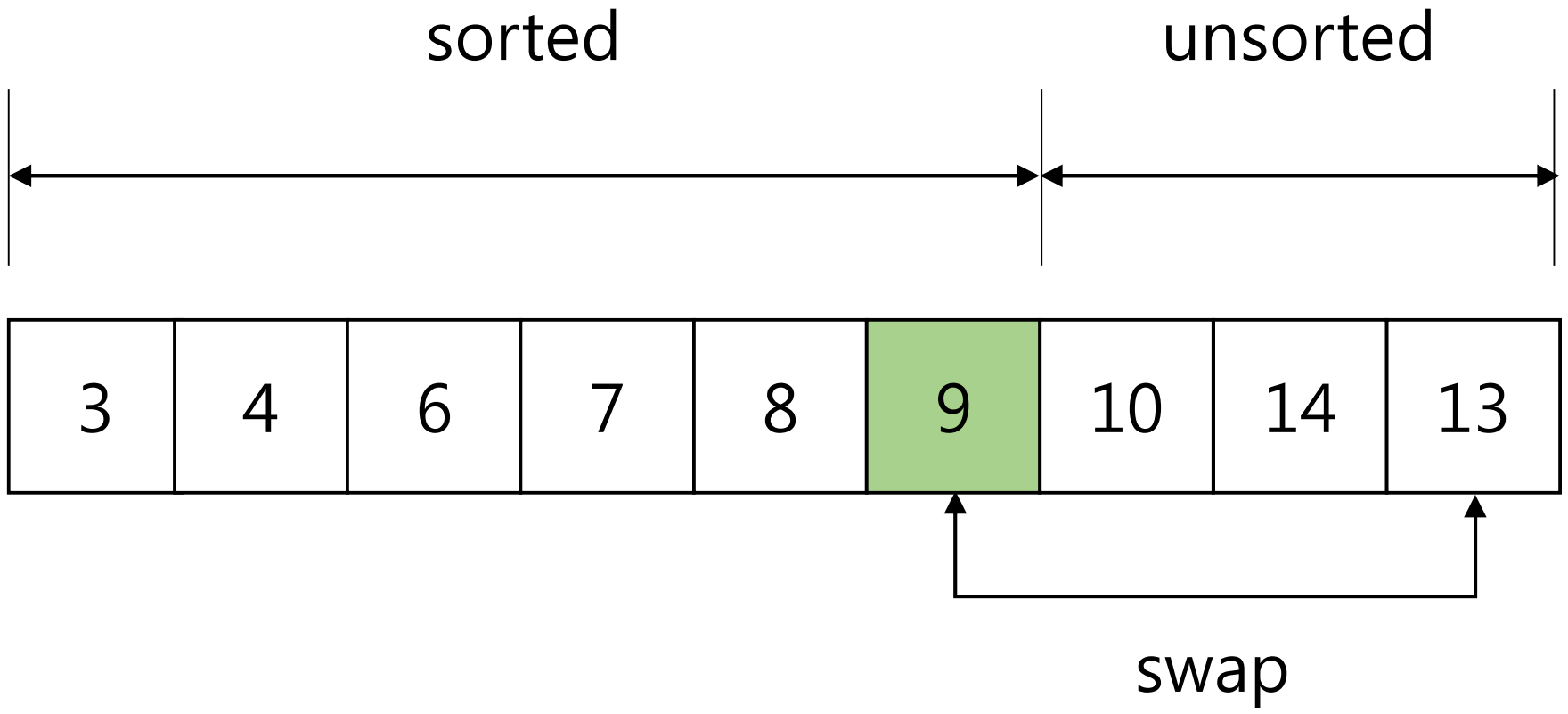
예시: 선택 정렬 (10/17)



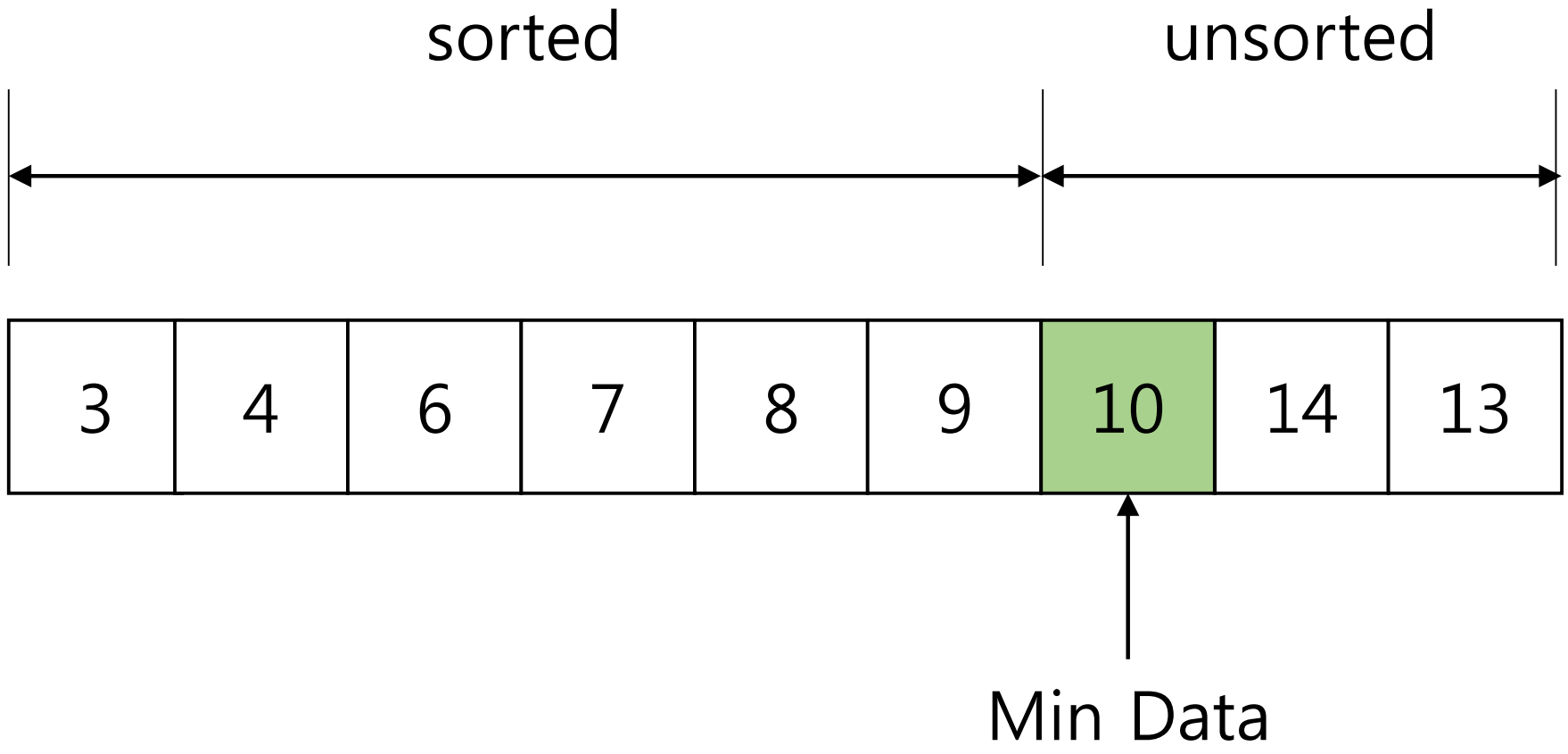
예시: 선택 정렬 (11/17)



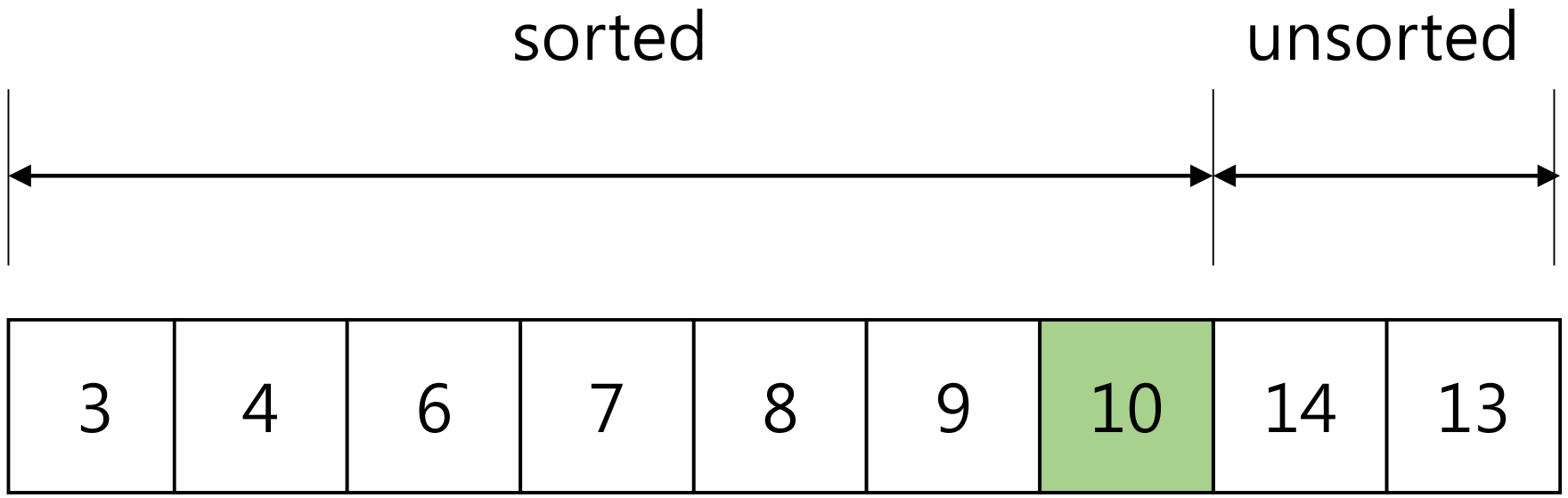
예시: 선택 정렬 (12/17)



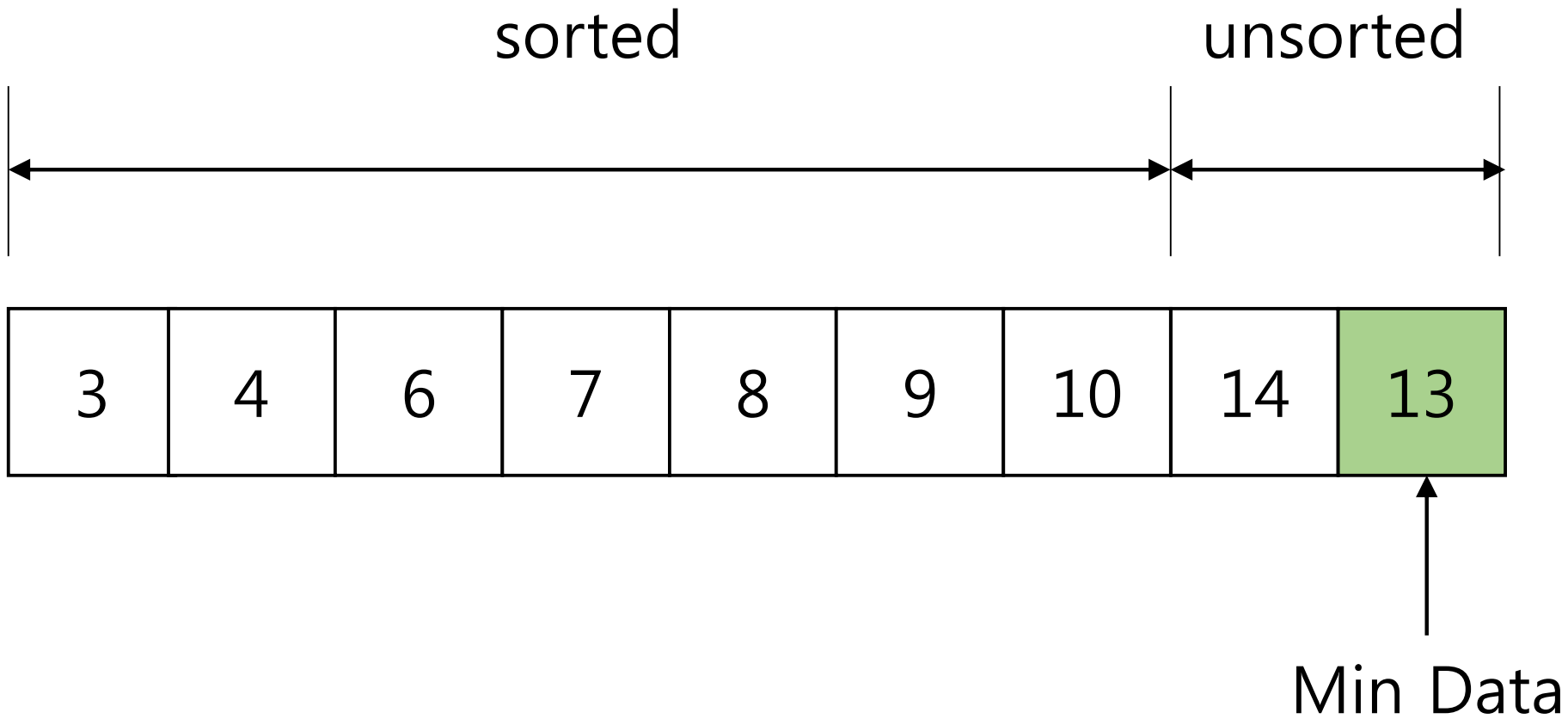
예시: 선택 정렬 (13/17)



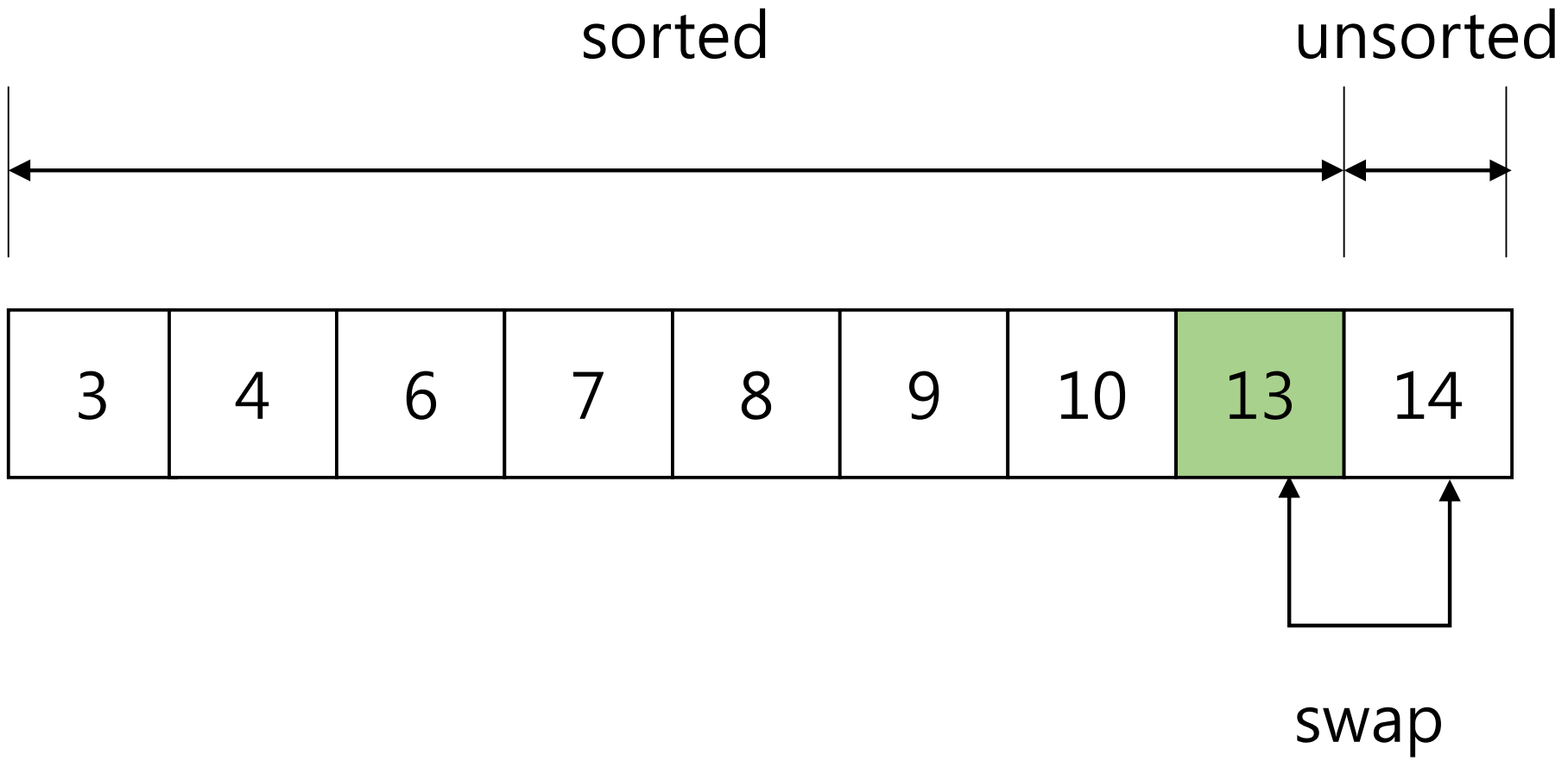
예시: 선택 정렬 (14/17)



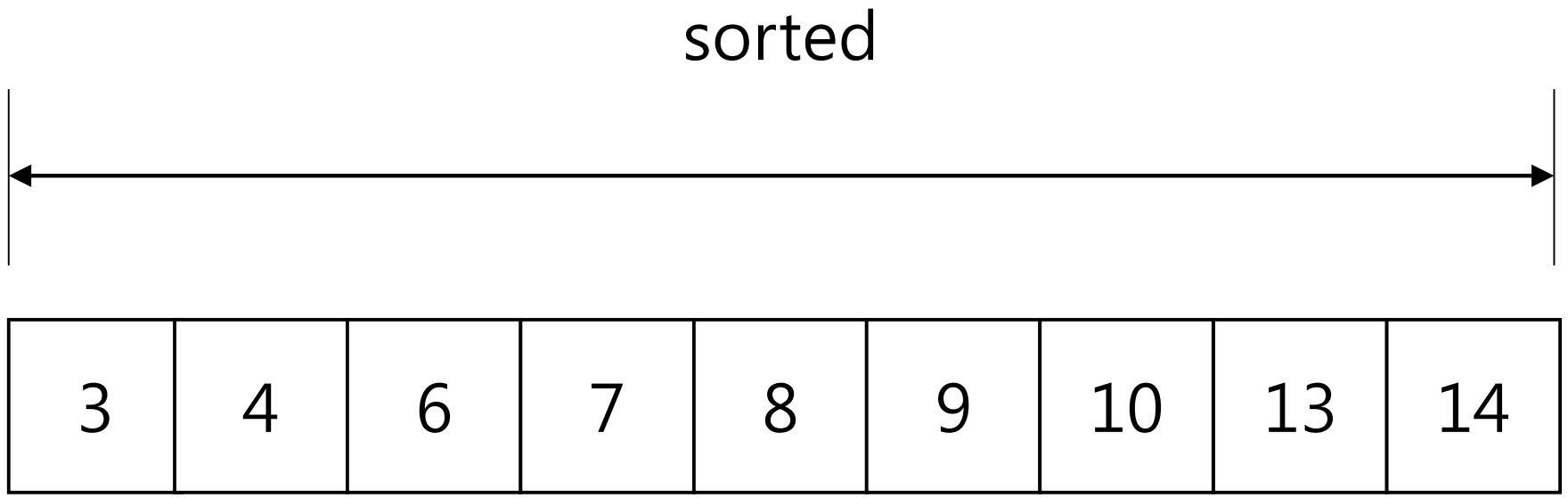
예시: 선택 정렬 (15/17)



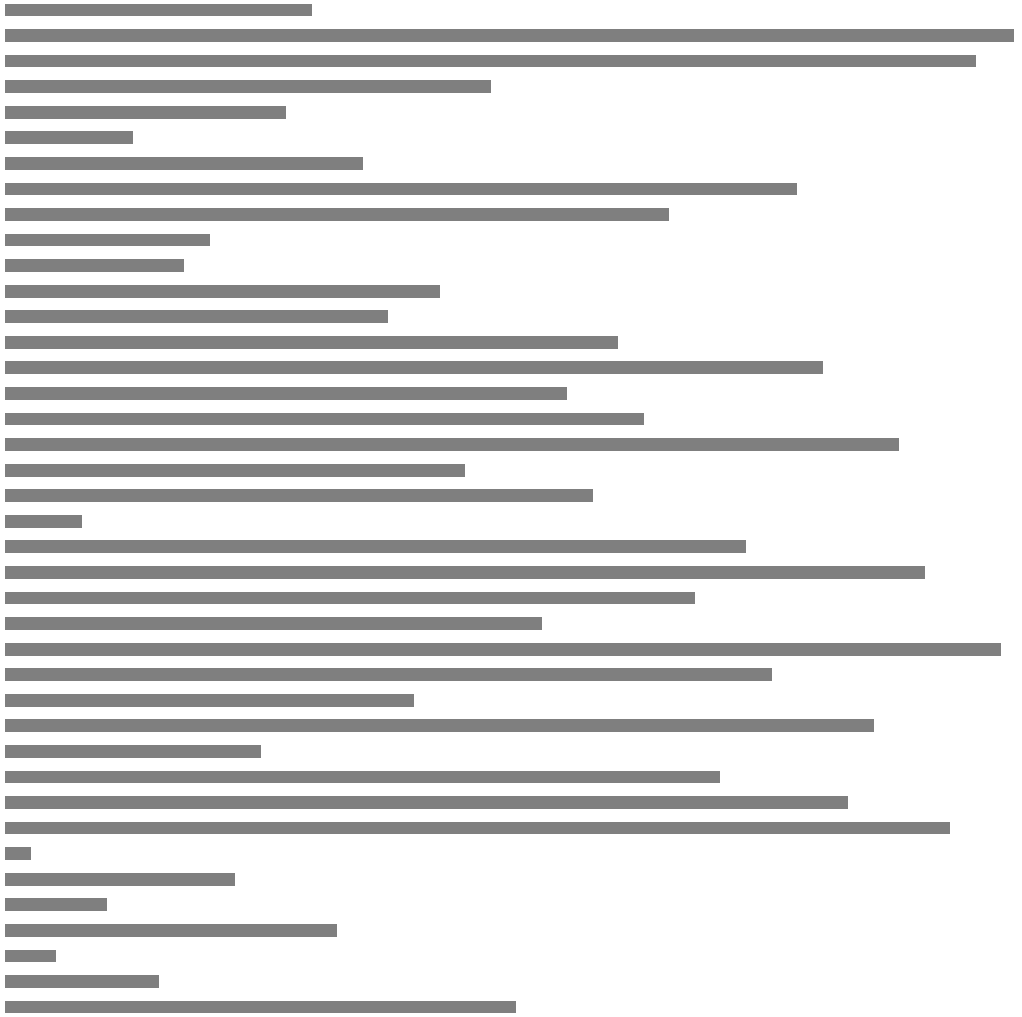
예시: 선택 정렬 (16/17)



예시: 선택 정렬 (17/17)



선택 정렬 실행 예시



selection_sort()

```
def selection_sort(t):
    for i in range(len(t) - 1):
        # find a location with the minimum item
        min_location = i
        for j in range(i + 1, len(t)):
            if t[min_location] > t[j]:
                min_location = j
        t[i], t[min_location] = t[min_location], t[i]  # swap items

t = [4, 3, 8, 6, 7, 13, 10, 14, 9]
selection_sort(t)
print(t)
```

```
[3, 4, 6, 7, 8, 9, 10, 13, 14]
```

선택 정렬 복잡도 분석

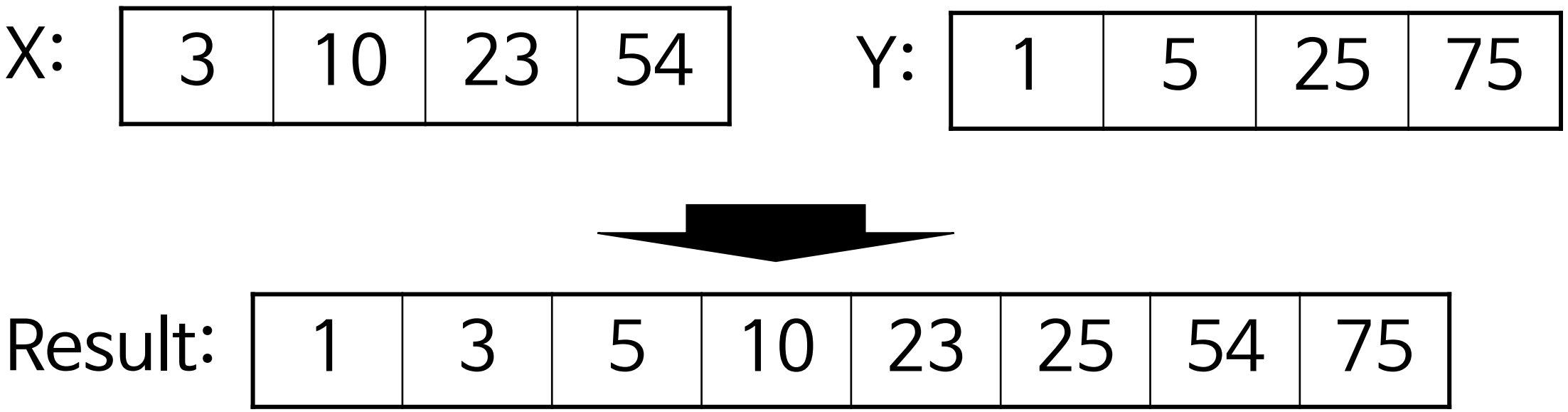
- 리스트에 n개의 아이템이 있을 경우, 다음 단계를 n-1회 반복
  - 정렬되지 않은 부분에서 가장 작은 수를 찾음
  - 가장 작은 수와 정렬되지 않은 부분의 첫번째 수를 교환
- 시간 복잡도
  - 비교회수: $(n-1)+(n-2)+\cdots+1=\frac{(n-1)\cdot n}{2}\rightarrow O(n^2)$
- 공간 복잡도
  - 교환을 위한 변수 1개 → $O(1)$

병합 정렬 (merge sort)

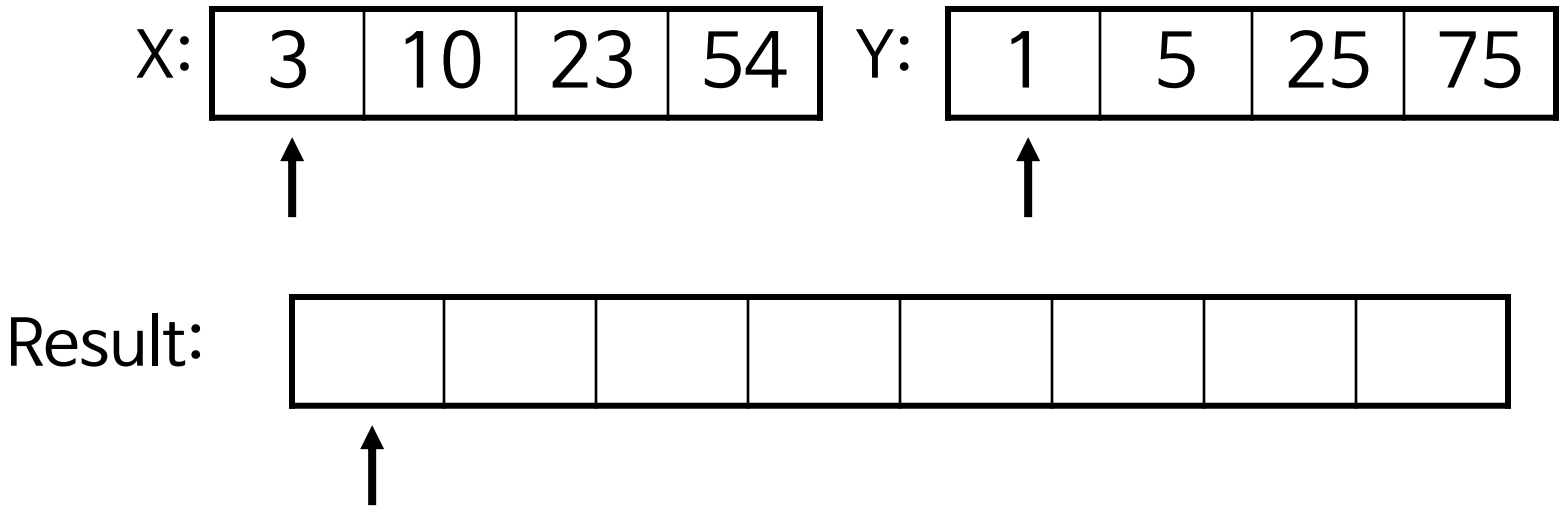
- 분할정복법에 의한 정렬 알고리즘
- n개의 아이템이 있는 리스트를 정렬하기 위하여,
  - n = 1: base case (리스트는 이미 정렬되어 있음)
  - n 〉 1: `merge_sort()`와 `merge()` 함수를 이용해 정렬
    - 왼쪽 절반을 `merge_sort()`를 이용해 정렬
    - 오른쪽 절반을 `merge_sort()`를 이용해 정렬
    - 정렬된 데이터를 `merge()`를 이용해 병합

병합 (merging)

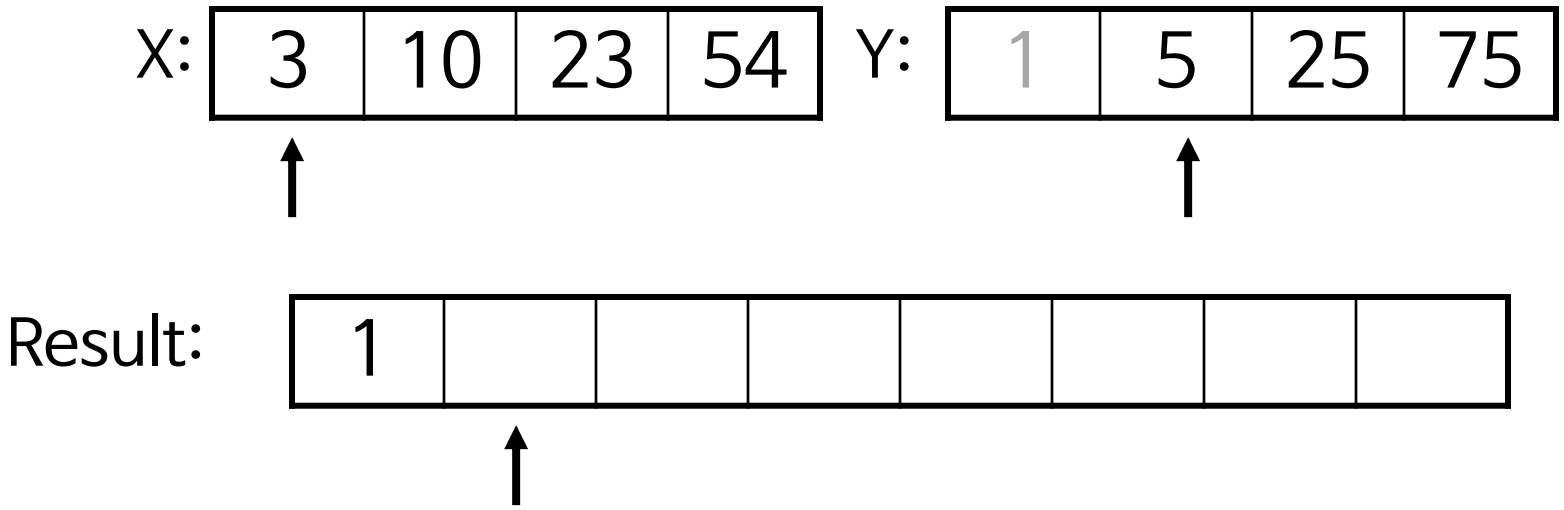
- 이미 정렬된 두 개의 리스트를 합쳐 하나의 정렬된 리스트로 만듦
- 예



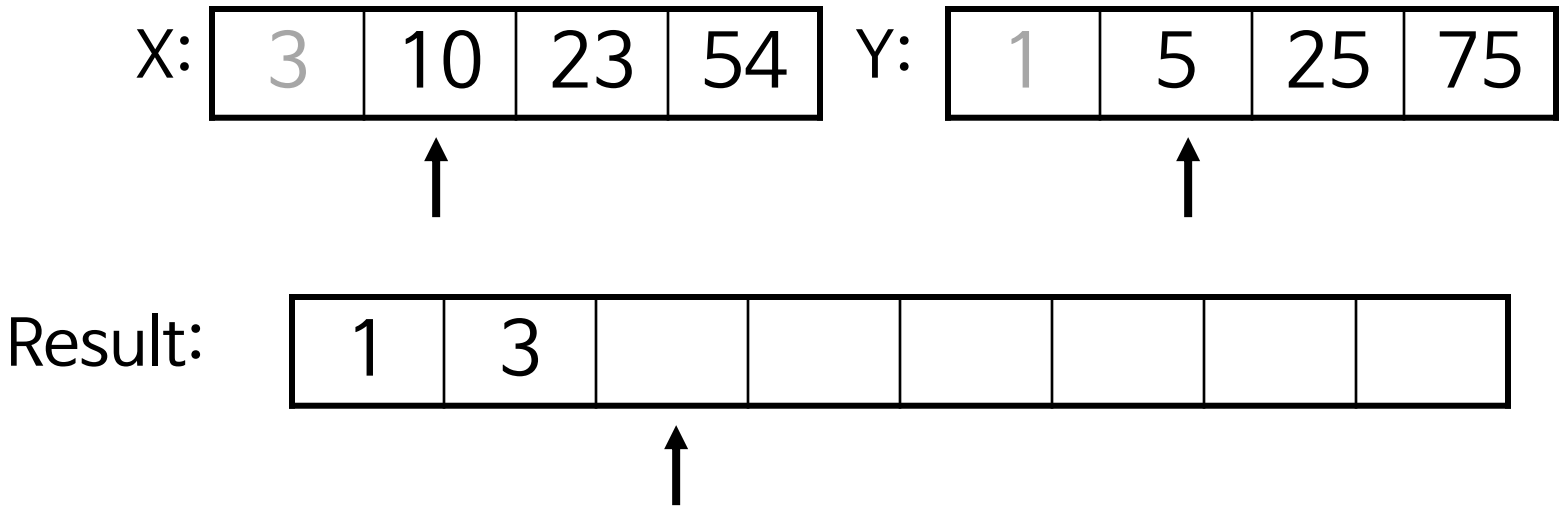
예시: 병합 (merging) (1/9)



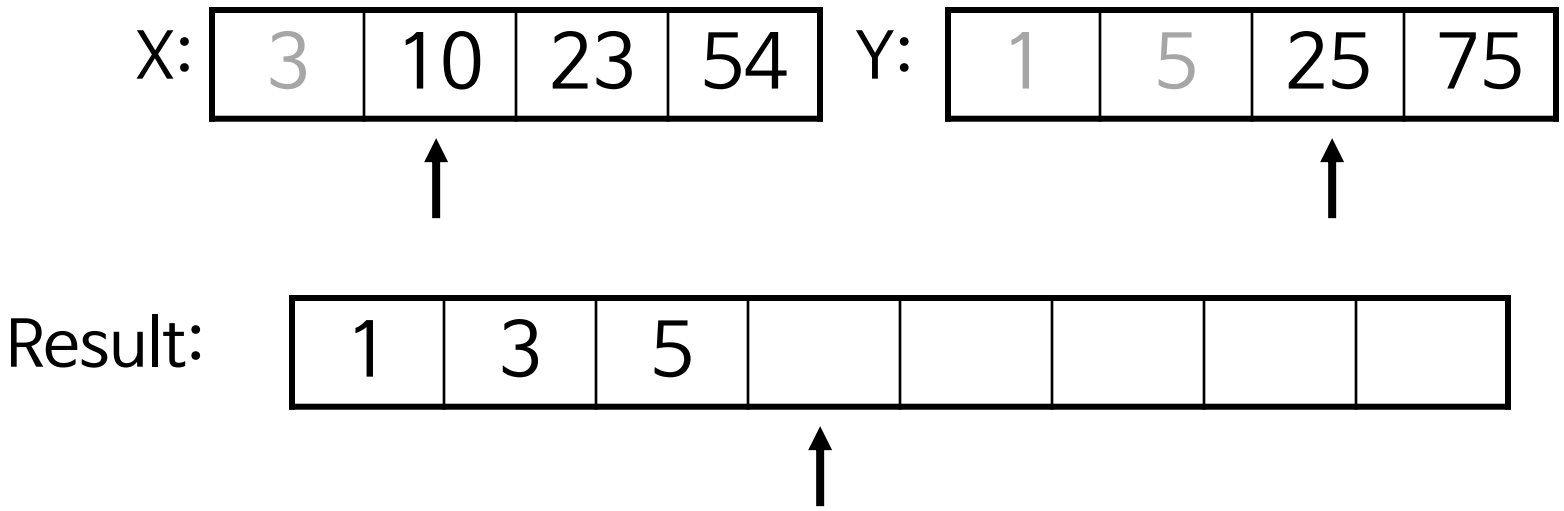
예시: 병합 (merging) (2/9)



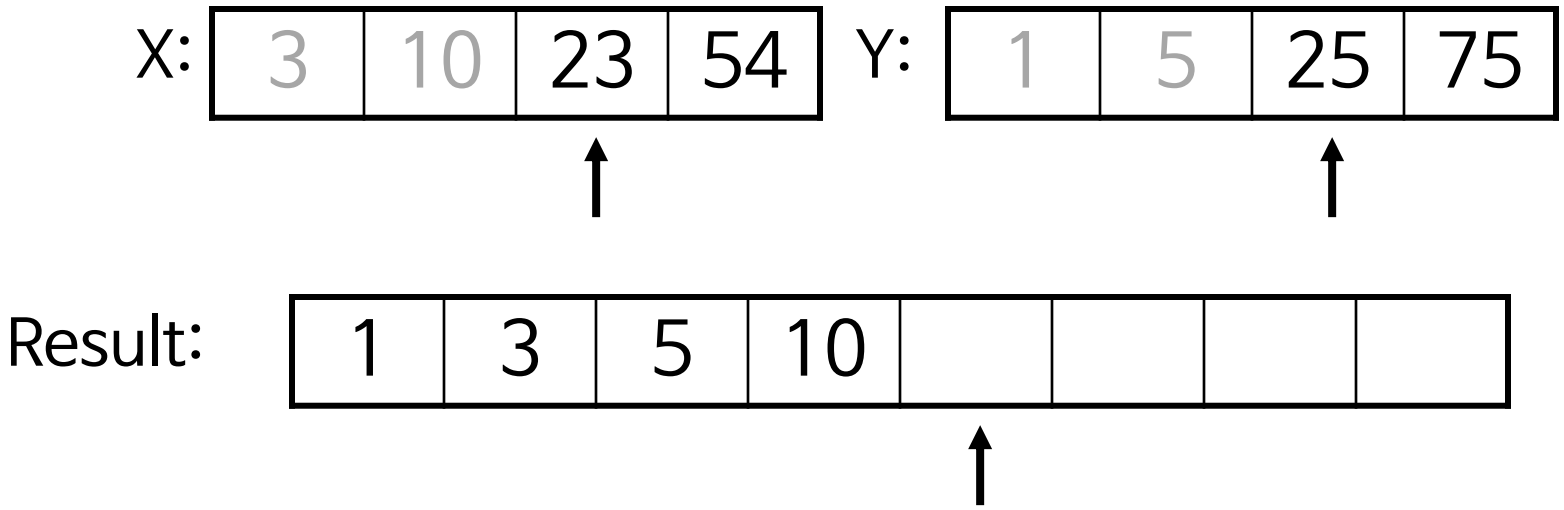
예시: 병합 (merging) (3/9)



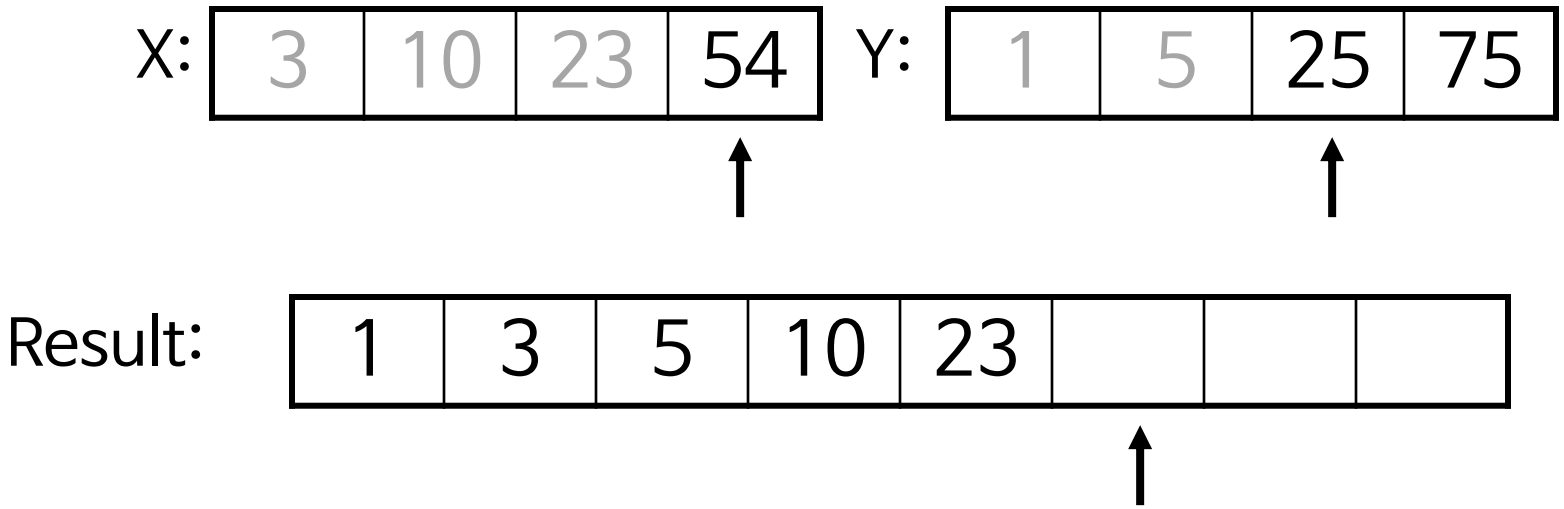
예시: 병합 (merging) (4/9)



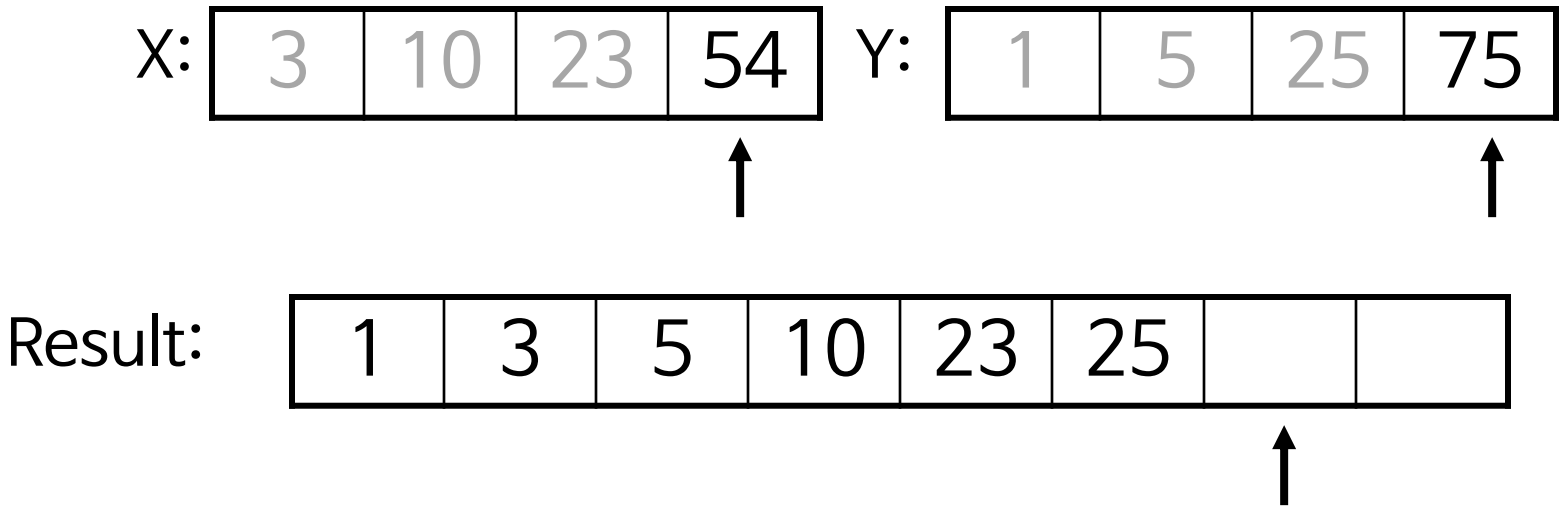
예시: 병합 (merging) (5/9)



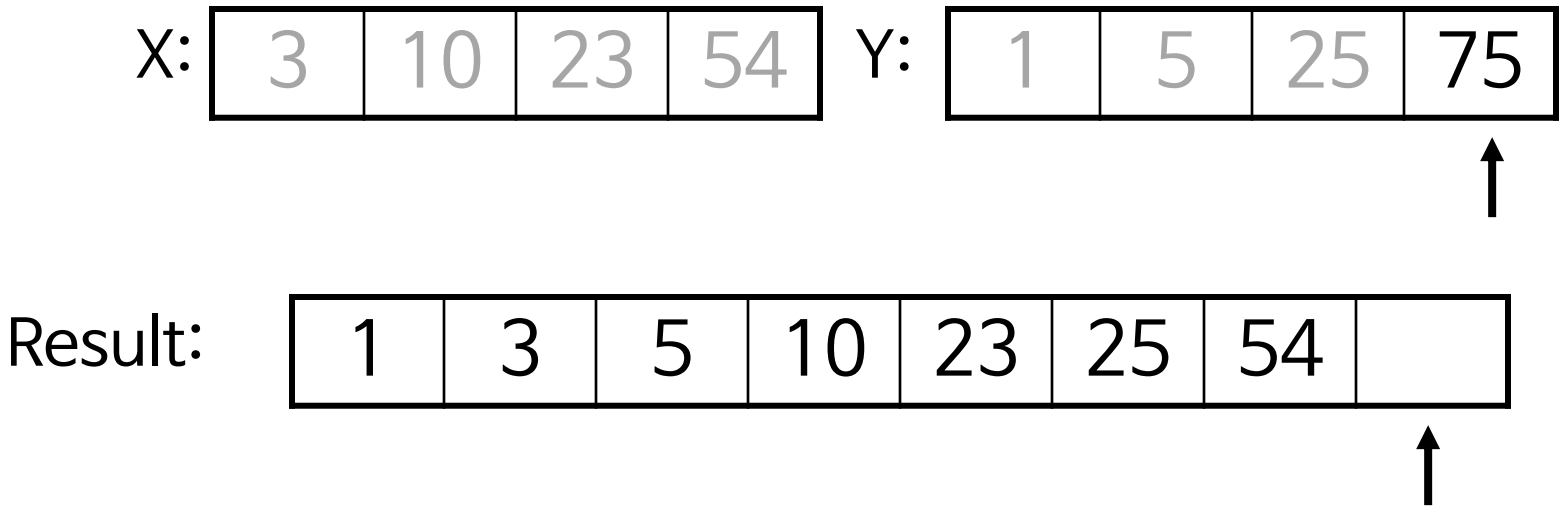
예시: 병합 (merging) (6/9)



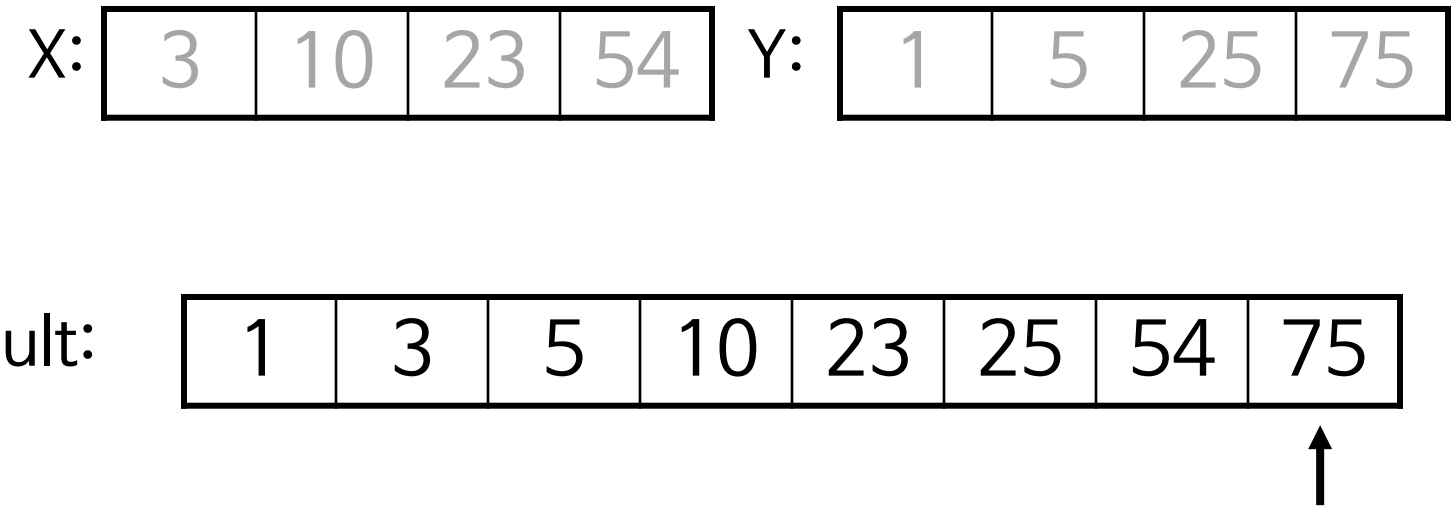
예시: 병합 (merging) (7/9)



예시: 병합 (merging) (8/9)



예시: 병합 (merging) (9/9)



예시: merge sort (1/10)

n = 9

99	6	86	15	58	35	86	4	0

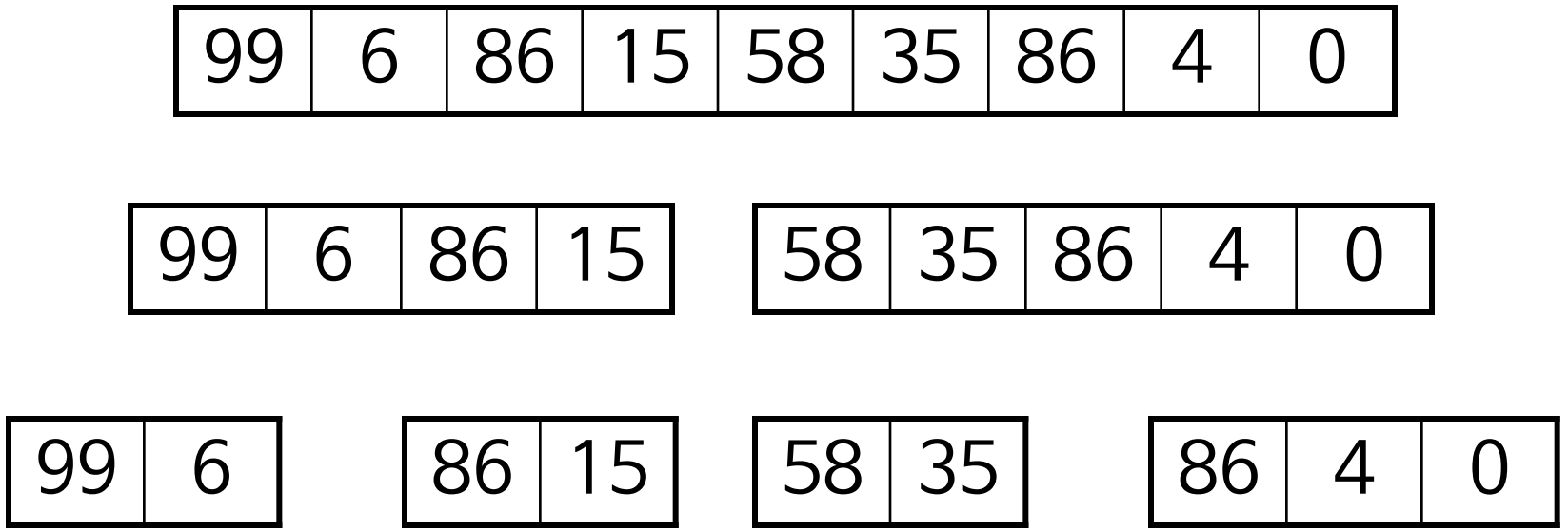
예시: merge sort (2/10)

99	6	86	15	58	35	86	4	0

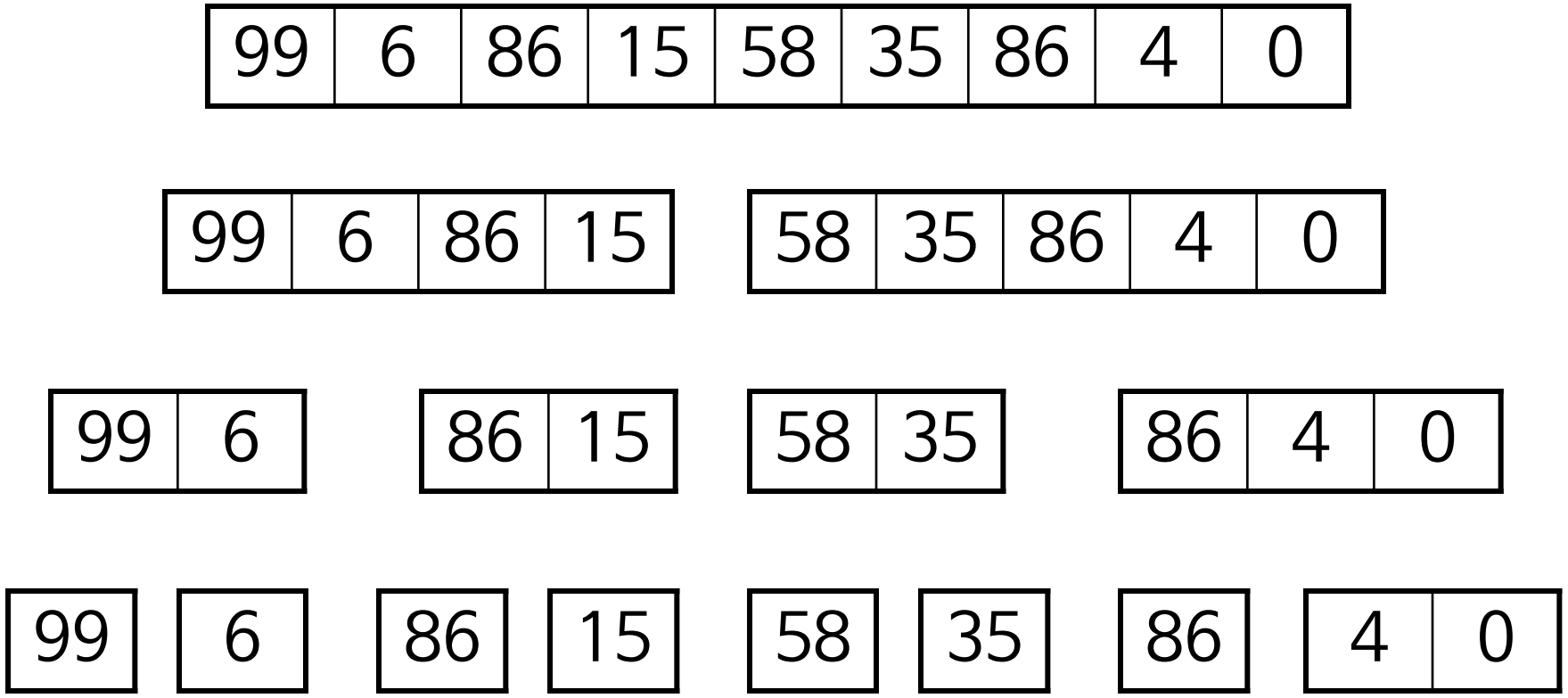
99	6	86	15

58	35	86	4	0

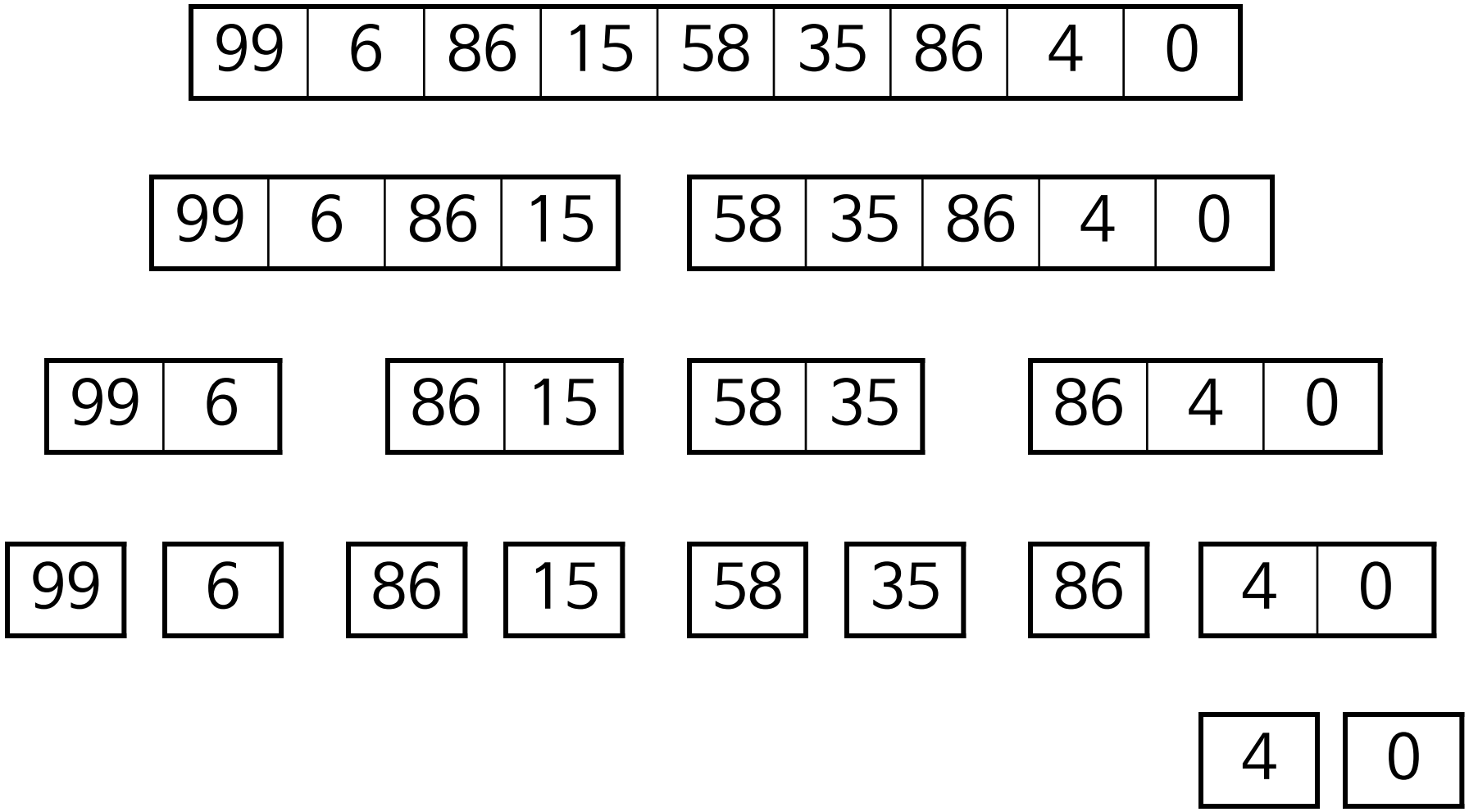
예시: merge sort (3/10)



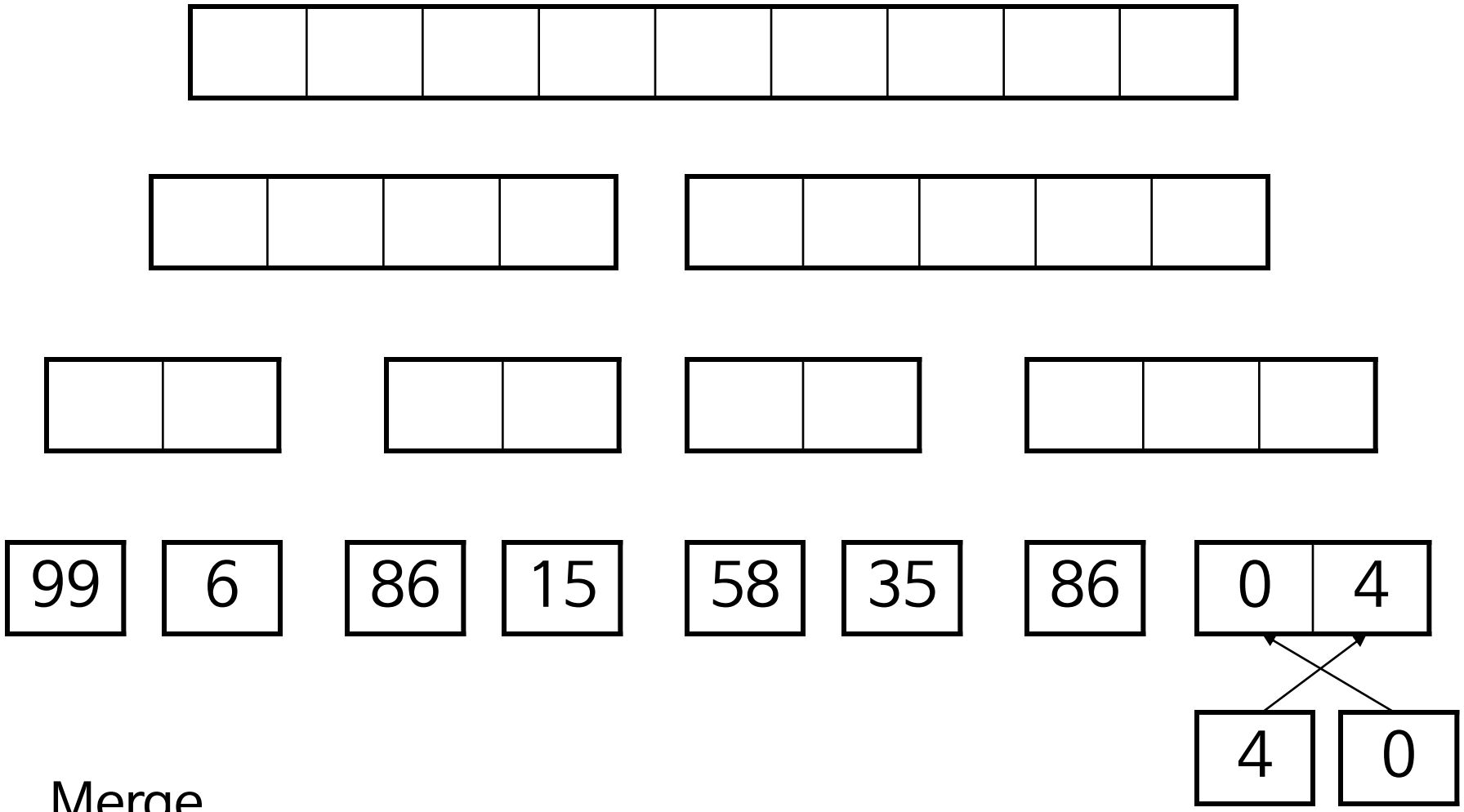
예시: merge sort (4/10)



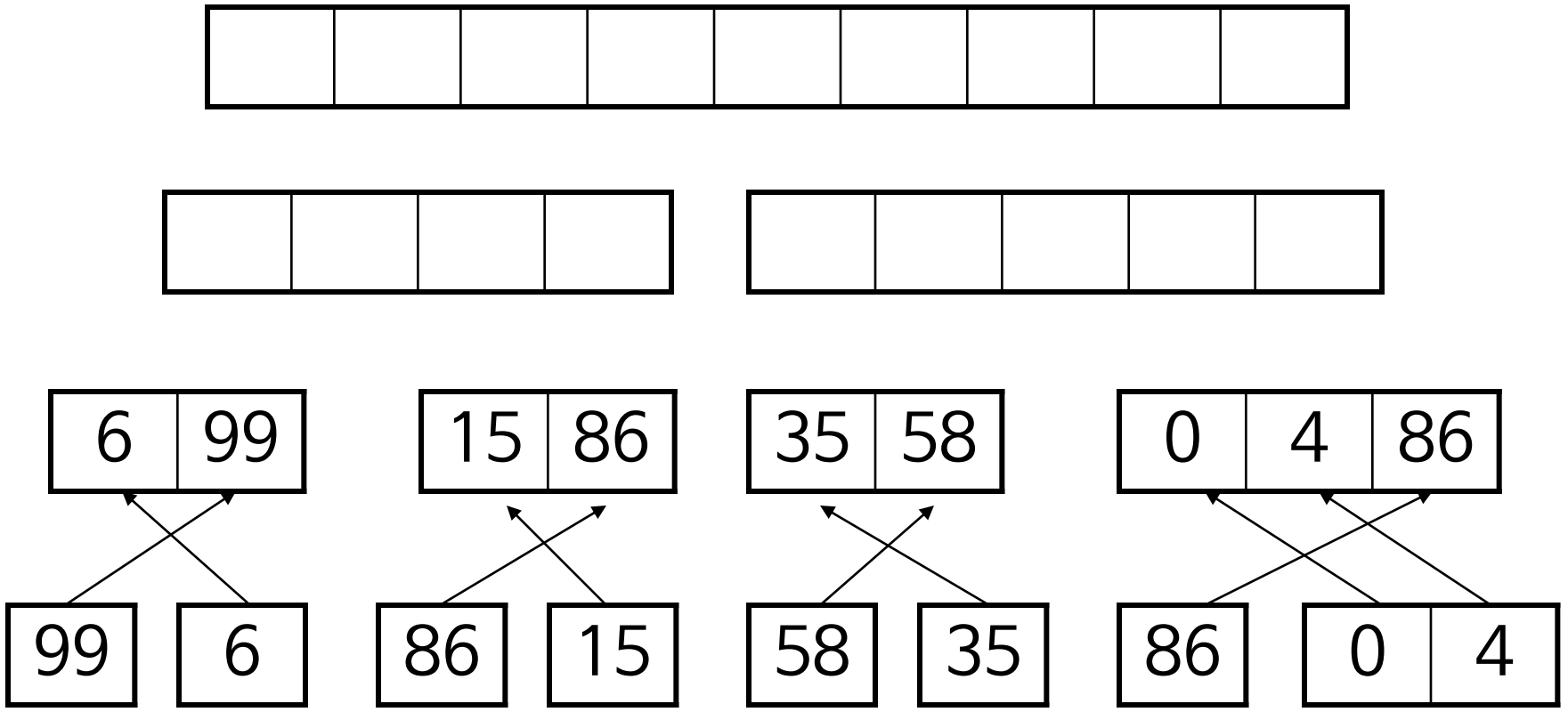
예시: merge sort (5/10)



예시: merge sort (6/10)

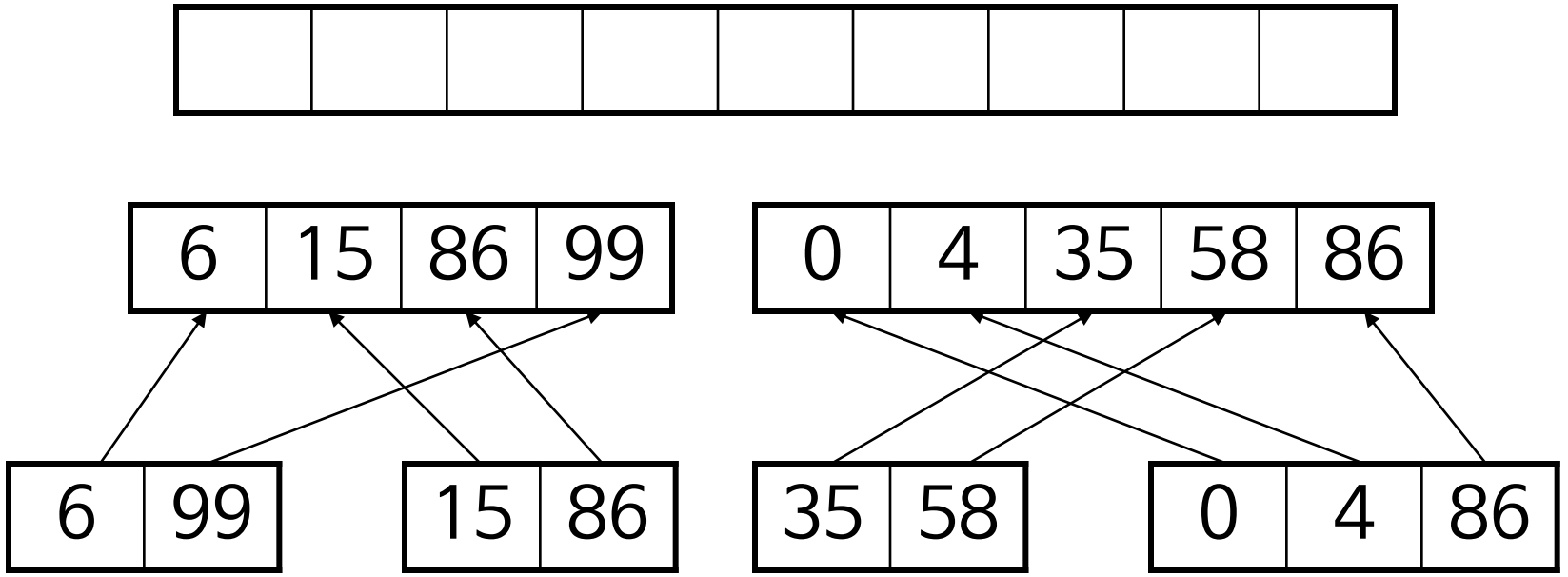


예시: merge sort (7/10)



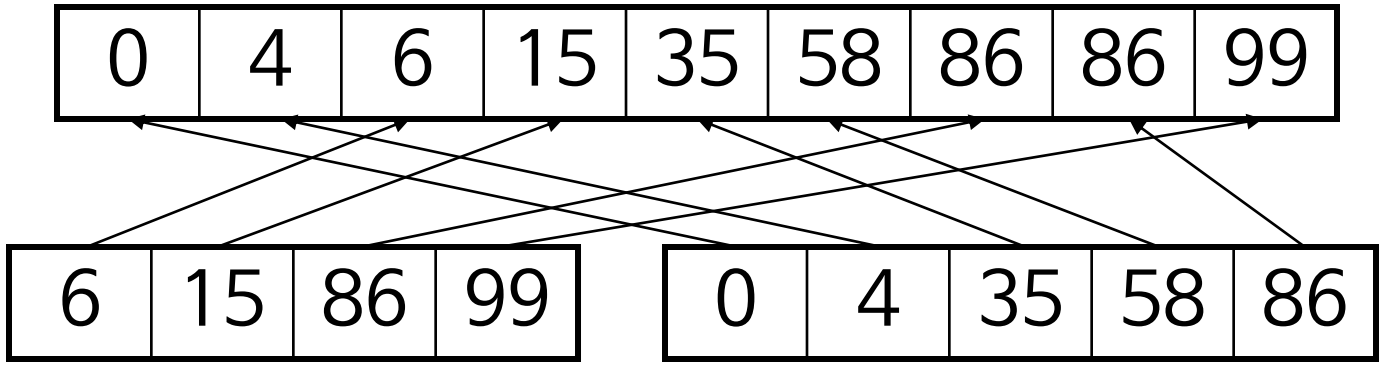
Merge

예시: merge sort (8/10)



Merge

예시: merge sort (9/10)

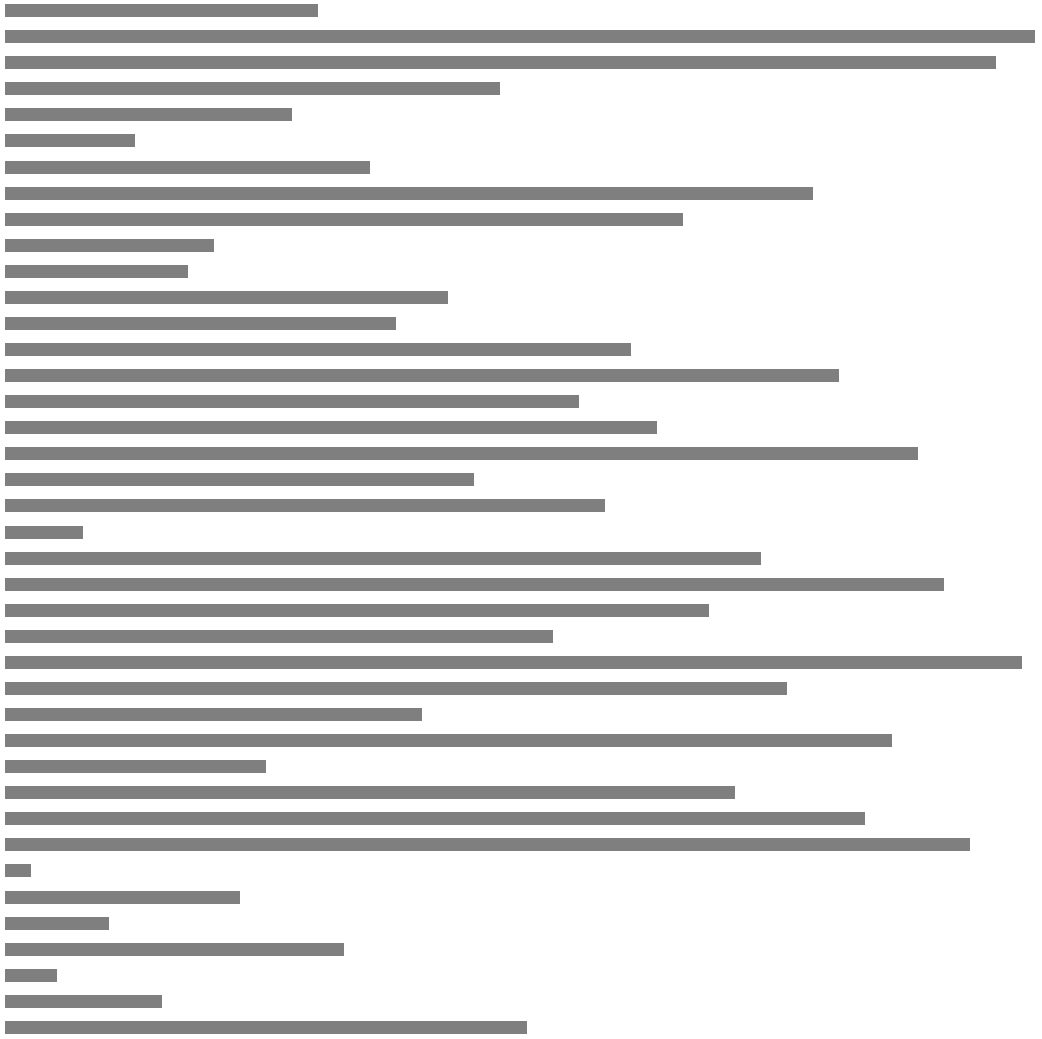


Merge

예시: merge sort (10/10)

0	4	6	15	35	58	86	86	99

병합 정렬 실행 예시



merge_sort() & merge()*

```
def merge(left, right):
    result = []
    i = 0
    j = 0
    while i < len(left) and j <
len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result
```

```
def merge_sort(t):
    if len(t) < 2:
        return t[:]
    mid = len(t) // 2
    left = merge_sort(t[:mid])
    right = merge_sort(t[mid:])
    return merge(left, right)

t = [99, 6, 86, 15, 58, 35, 86, 4, 0]
t = merge_sort(t)
print(t)
```

```
[0, 4, 6, 15, 35, 58, 86, 86, 99]
```

* sorted()처럼 인자로 넘겨진 리스트는 변경하지 않고, 별도의 정렬된 리스트를 반환

참고 (다른 구현*): merge_sort()

```
def merge_sort(t, l, r):
    # base case
    if l >= r:
        return

    # recursive case
    mid = (l + r) // 2
    merge_sort(t, l, mid)
    merge_sort(t, mid + 1, r)
    merge(t, l, r)

t = [99, 6, 86, 15, 58, 35, 86, 4,
0]
merge_sort(t, 0, len(t) - 1)
print(t)
```

```
[0, 4, 6, 15, 35, 58, 86, 86, 99]
```

* sort()처럼, 리스트 안의 원소들끼리 위치를 변경하여 정렬

참고 (다른 구현): merge() (cont.)

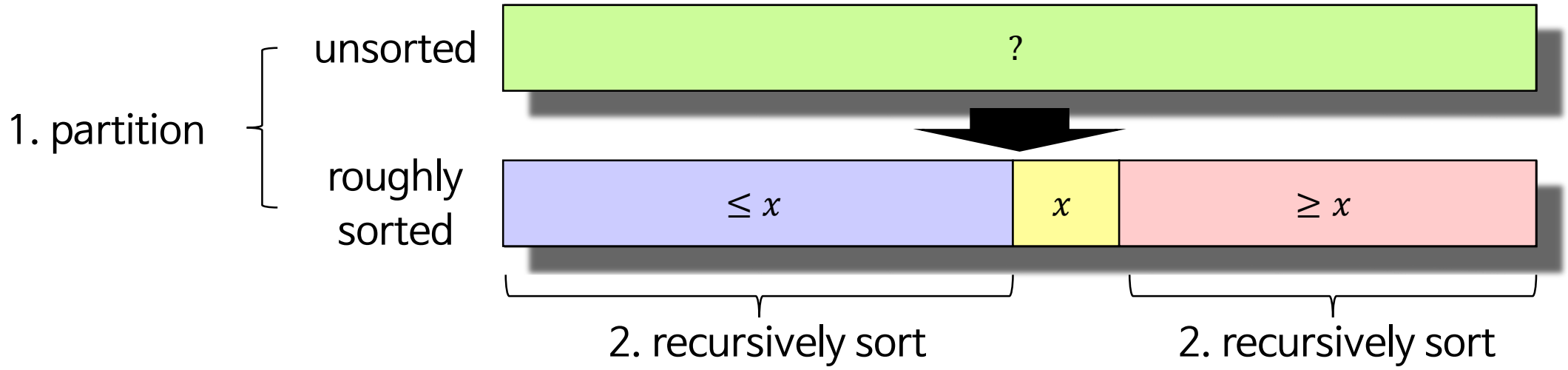
```
def merge(t, l, r):
    result = []
    mid = (l + r) // 2
    lptr = l
    rptr = mid + 1
    # compare the values from the
left
    # half and right half, then add
    # the smaller value to the
result
    while lptr <= mid and rptr <= r:
        if t[lptr] < t[rptr]:
            result.append(t[lptr])
            lptr += 1
        else:
            result.append(t[rptr])
            rptr += 1
    # add remaining items from the
    # lefthalf
    while lptr <= mid:
        result.append(t[lptr])
        lptr += 1

    # add remaining items from the
    # righthalf
    while rptr <= r:
        result.append(t[rptr])
        rptr += 1

    # update the original list
    for i in range(len(result)):
        t[l + i] = result[i]
```

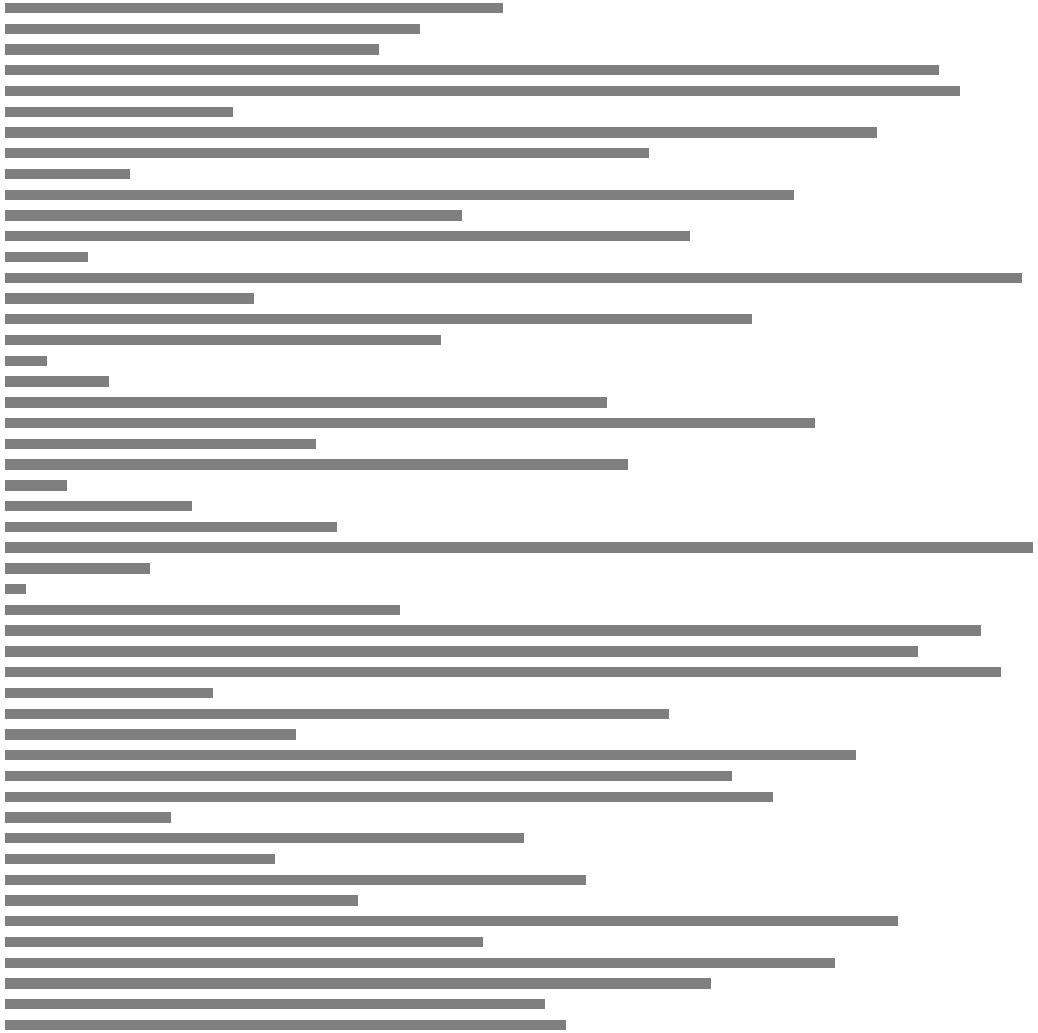
퀵 정렬 (quick sort)

- 평균적으로 가장 빠른* 정렬 알고리즘으로 가장 빈번하게 사용됨
  – merge sort, heap sort에 비해 평균 2-3배 빠름
- 분할정복법을 이용 (일반적으로 재귀를 사용하여 구현됨)
  1. partition: pivot value보다 큰 수와 작은 수의 영역으로 분할
     - pivot value는 일반적으로 가장 앞, 가운데, 끝의 값 중 하나를 사용
  2. 두 개로 분할된 영역을 재귀적으로 정렬



* 실제 실행시간이 가장 빠른 알고리즘 중 하나

퀵 정렬 실행 예시



퀵 소트 구현*: 파티션 별로 리스트를 만들고 원소를 복사

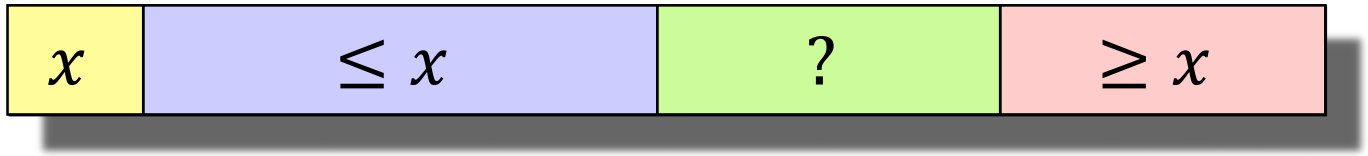
```
def quicksort(t):
    # base case
    if len(t) <= 1:
        return t[:]
    # take the value in the middle as a pivot value
    pivot = t[len(t) // 2]
    less = []
    more = []
    equal = []
    for a in t:
        if a < pivot:
            less.append(a)
        elif a > pivot:
            more.append(a)
        else:
            equal.append(a)
    return quicksort(less) + equal + quicksort(more)
```

* sorted()처럼 인자로 넘겨진 리스트는 변경하지 않고, 별도의 정렬된 리스트를 반환

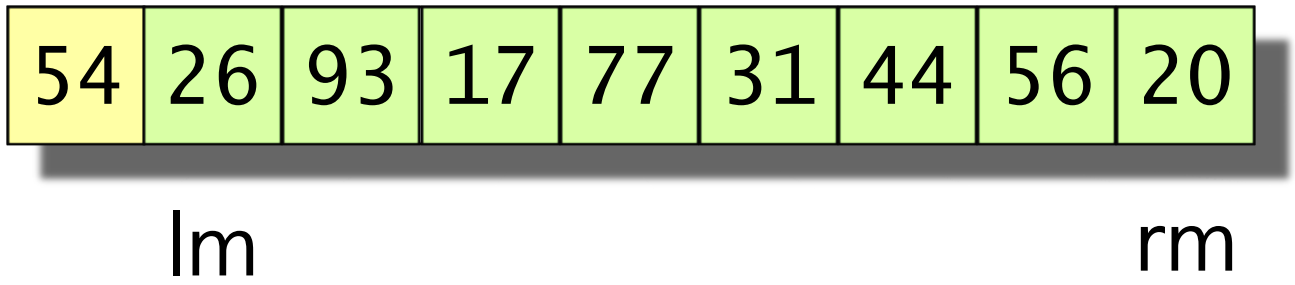
파티션 방법 1

- 첫 번째 수를 pivot value(=x)로 정하고,
  (첫 번째 수를 제외한) 왼쪽 끝부터 $\leq x$인 수를,
  오른쪽 끝부터 $\geq x$수가 되도록 아래 과정을 반복

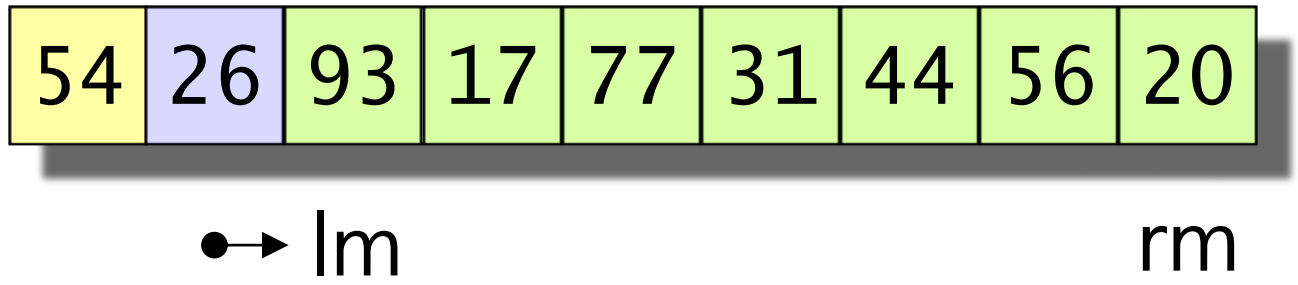
  1. 첫 번째 수를 pivot value로 지정
  2. leftmark(= lm)는 두 번째 수의 위치(리스트의 인덱스),
     rightmark(=rm)는 가장 오른쪽 수의 위치로 지정
  3. $lm \leq rm$이고, $t[lm] \leq x$이면 lm을 1증가 (오른쪽으로 이동)
  4. $rm \geq lm$이고, $t[rm] \geq x$이면 rm를 1감소 (왼쪽으로 이동)
  5. $lm < rm$이면, lm과 rm 위치의 수를 교환하고 3번부터 반복
  6. 가장 왼쪽의 수(즉, pivot value)와 rm 위치의 수를 교환



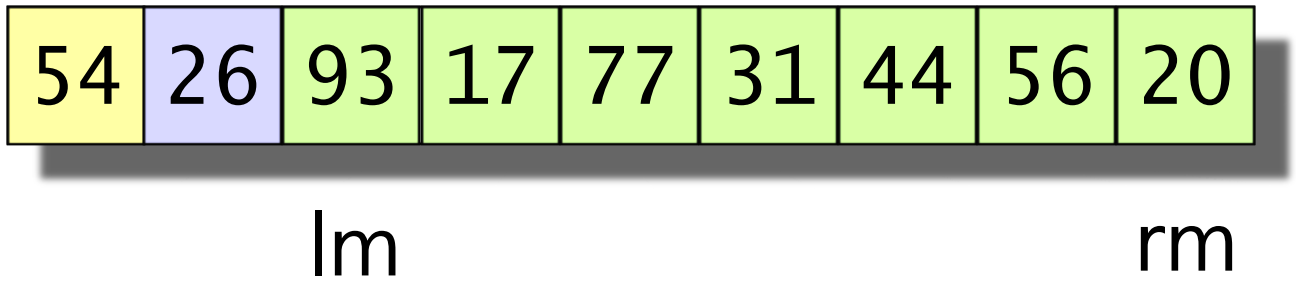
예시: 파티션 방법 1 (1/13)



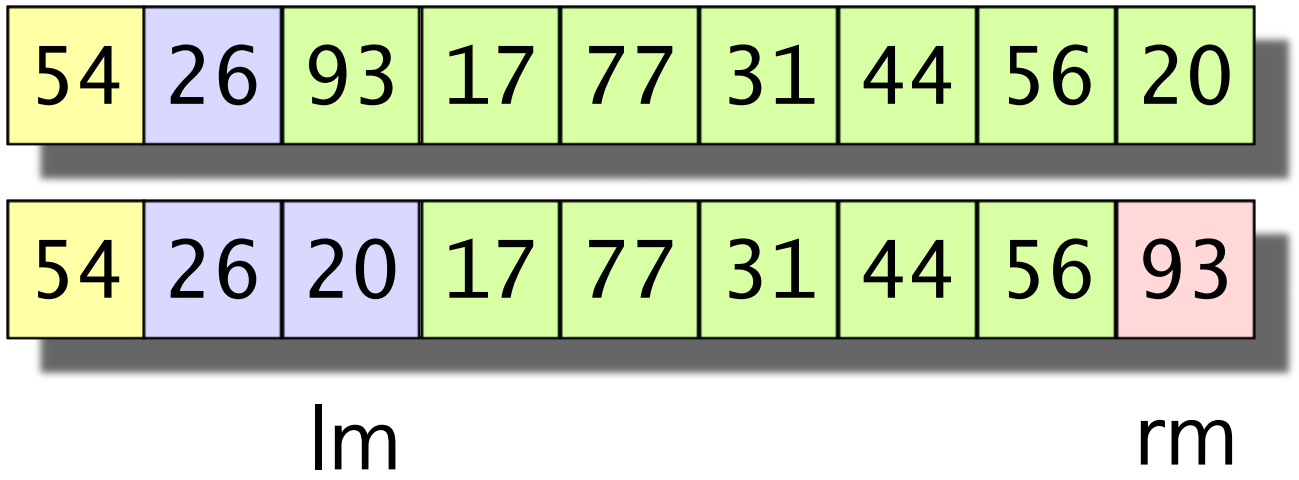
예시: 파티션 방법 1 (2/13)



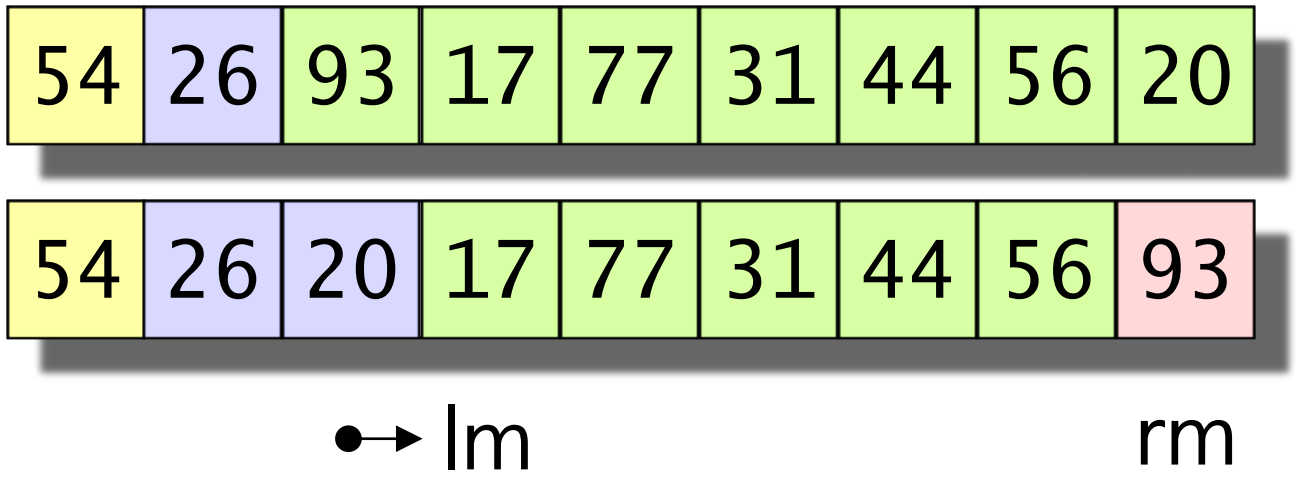
예시: 파티션 방법 1 (3/13)



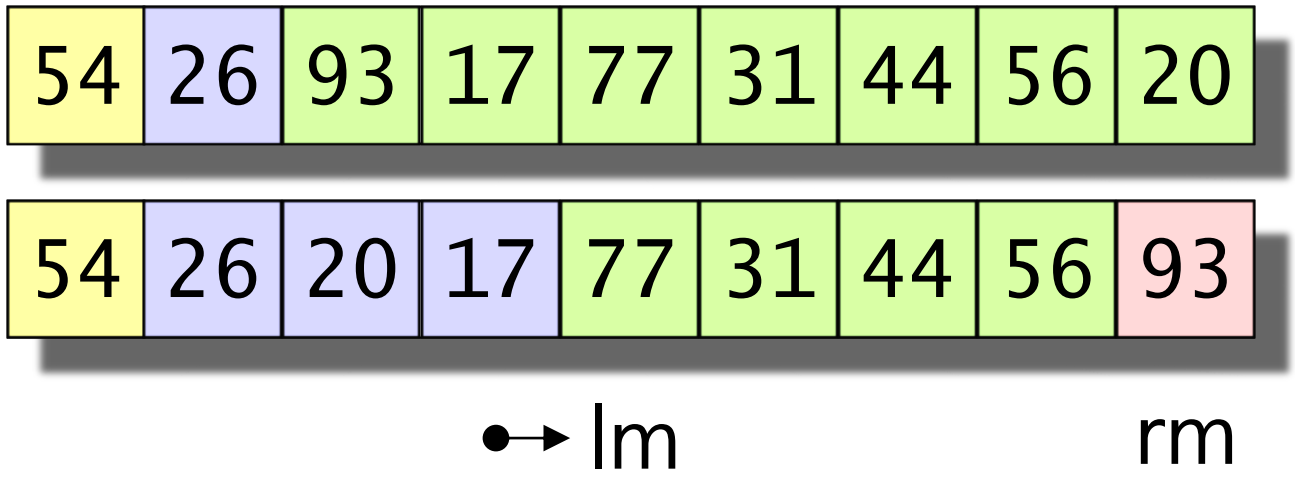
예시: 파티션 방법 1 (4/13)



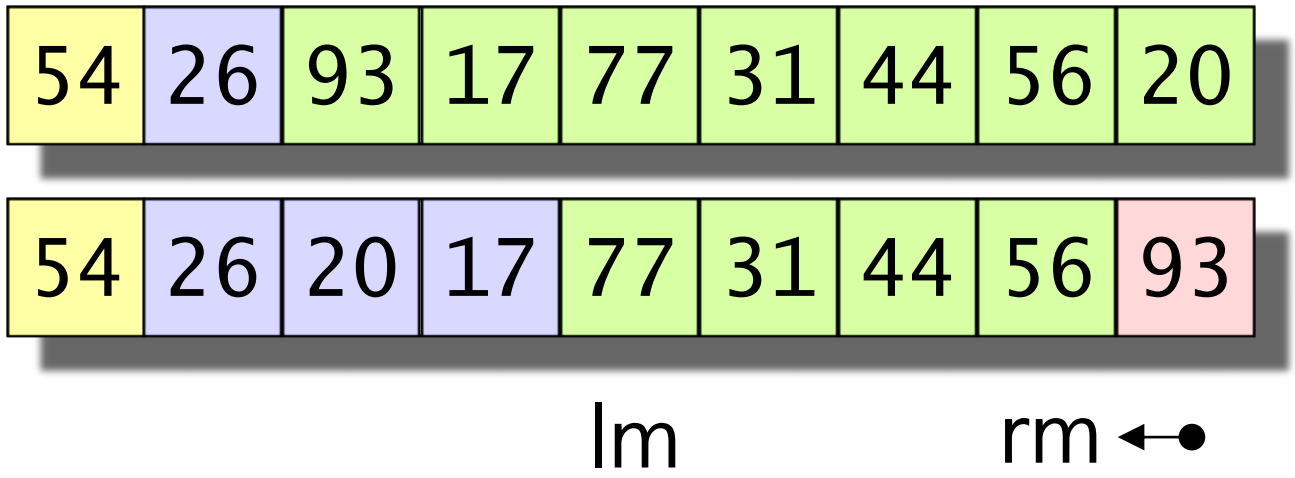
예시: 파티션 방법 1 (5/13)



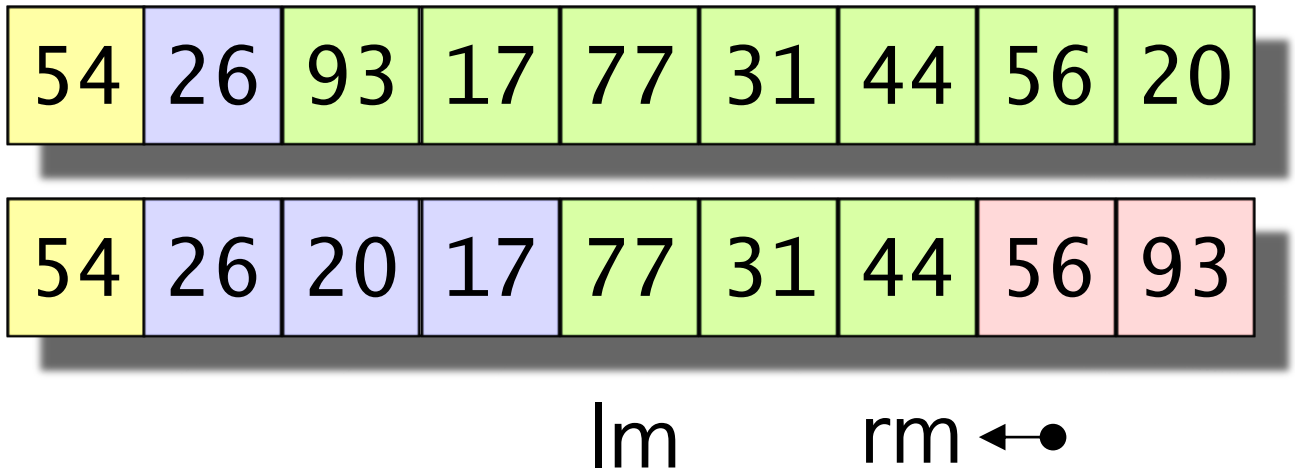
예시: 파티션 방법 1 (6/13)



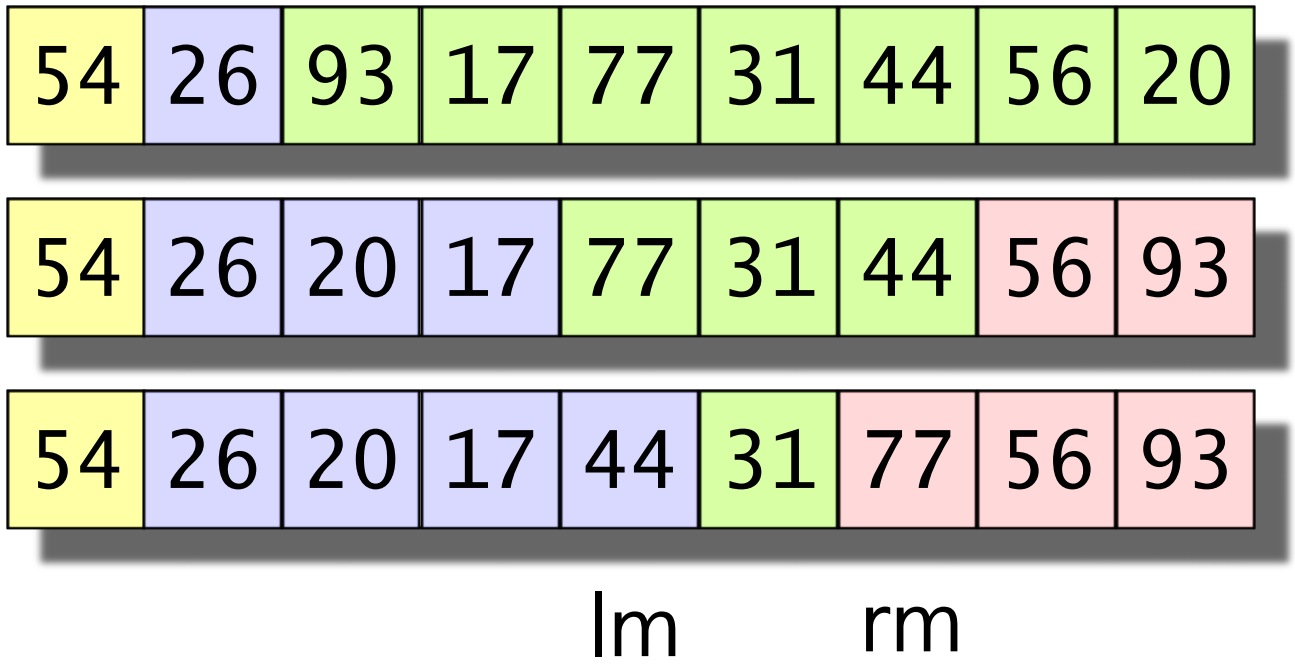
예시: 파티션 방법 1 (7/13)



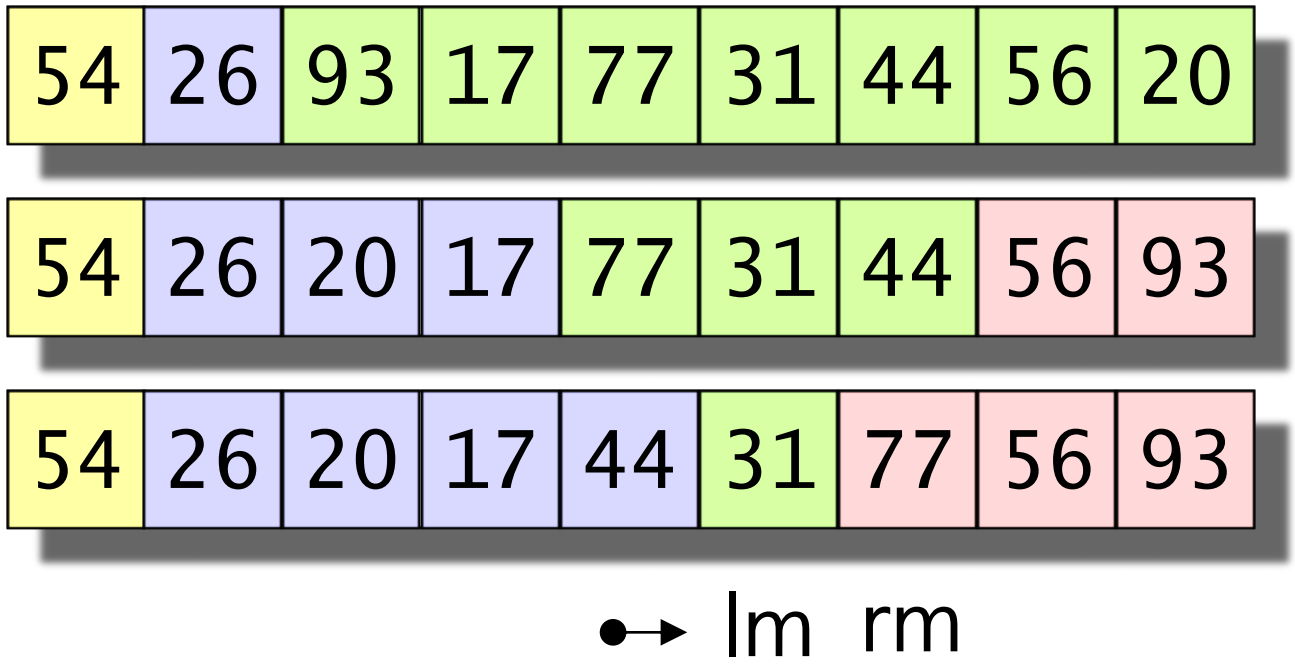
예시: 파티션 방법 1 (8/13)



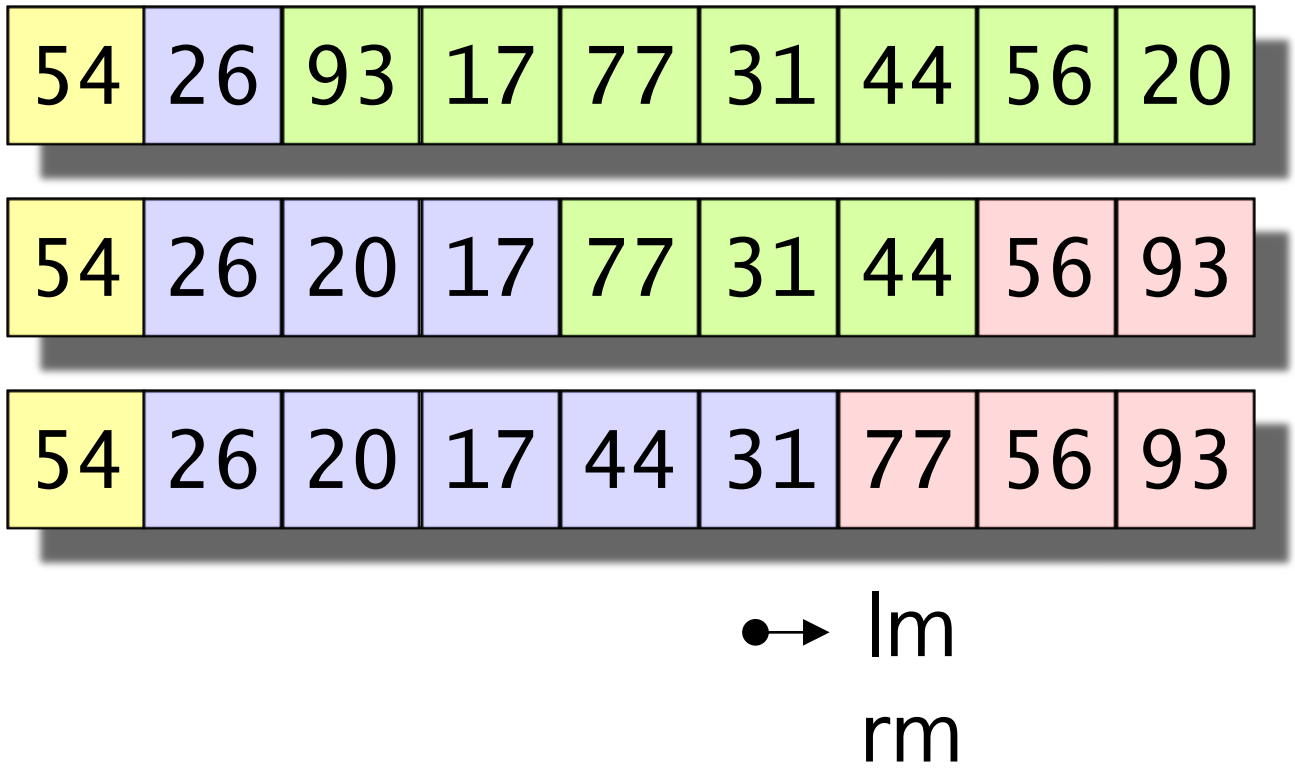
예시: 파티션 방법 1 (9/13)



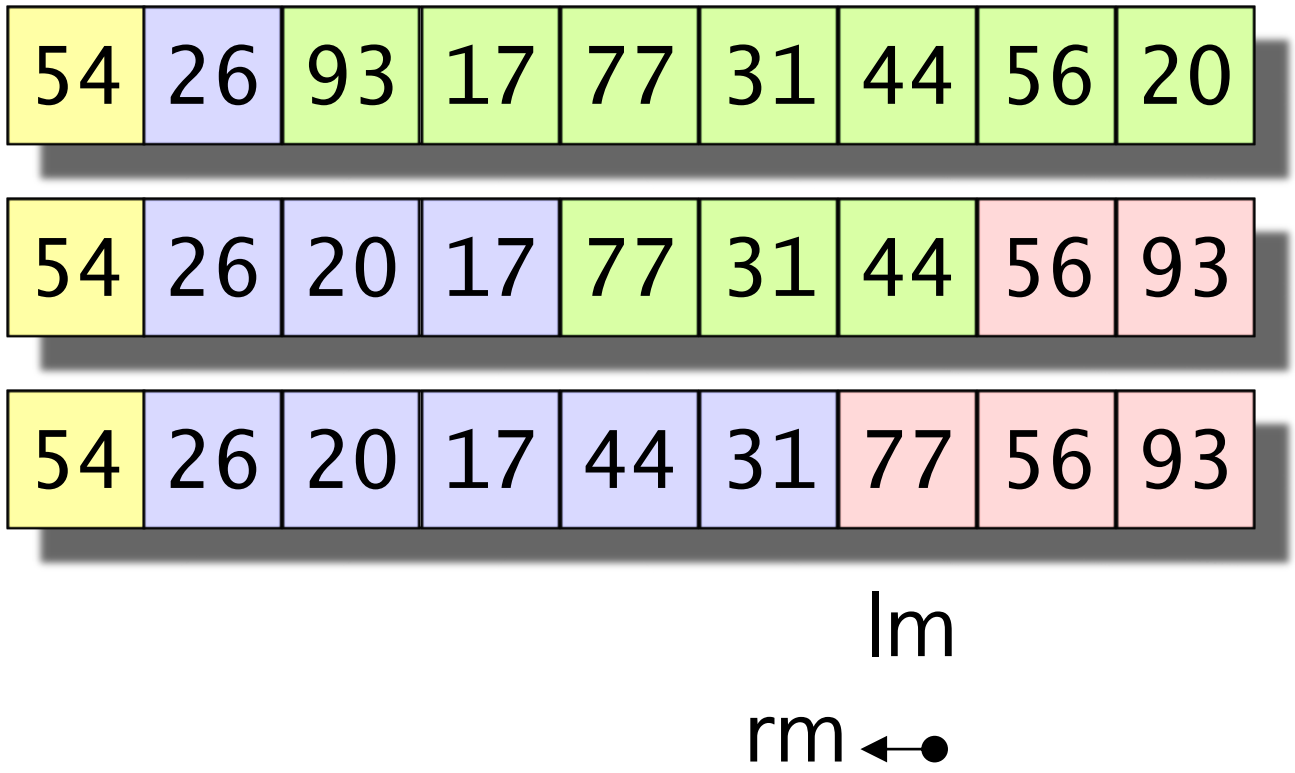
예시: 파티션 방법 1 (10/13)



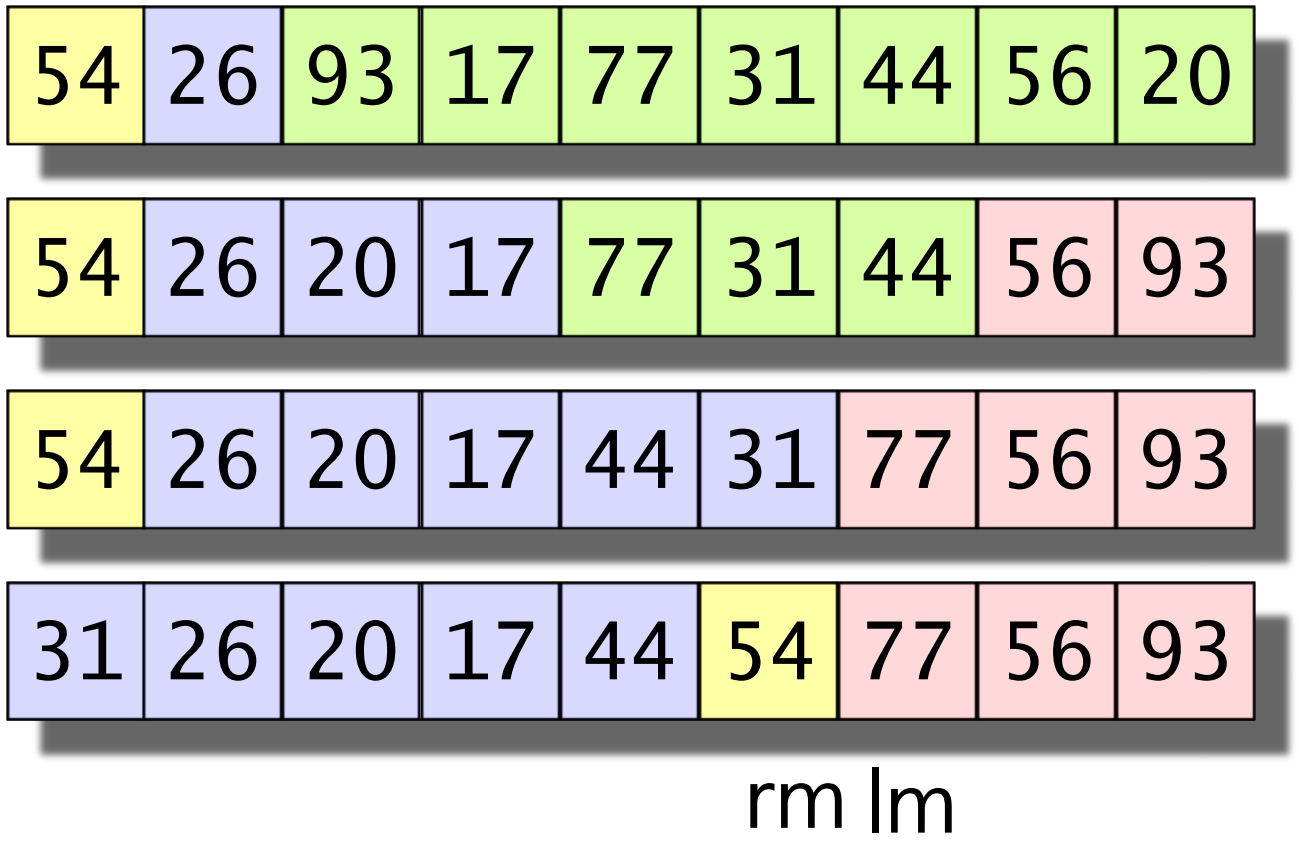
예시: 파티션 방법 1 (11/13)



예시: 파티션 방법 1 (12/13)



예시: 파티션 방법 1 (13/13)



파티션 방법 2

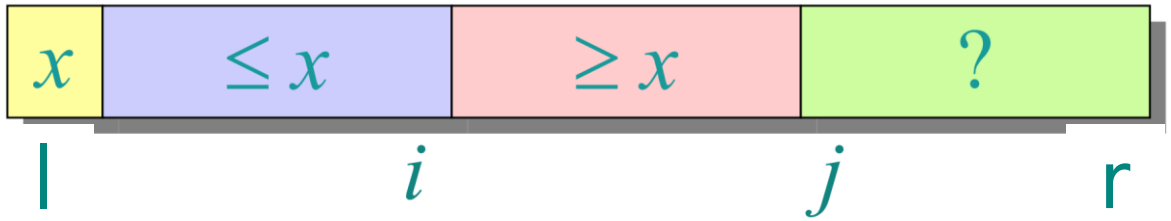
- 첫 번째 수를 pivot value(=x)로 정하고,
  j를 l(가장 왼쪽의 인덱스)부터 r(가장 오른쪽의 인덱스)까지 증가시키면서,
  k번째 위치(인덱스)에 있는 수들이 다음을 만족하도록, 아래 1-3번의 과정을 수행
  - $k \leq i$: $\leq x$
  - $i < k \leq j$: $\geq x$
  - $k > j$: $unknown$ (?)

  1.첫 번째 수를 pivot value로 지정하고, i에 l을 대입

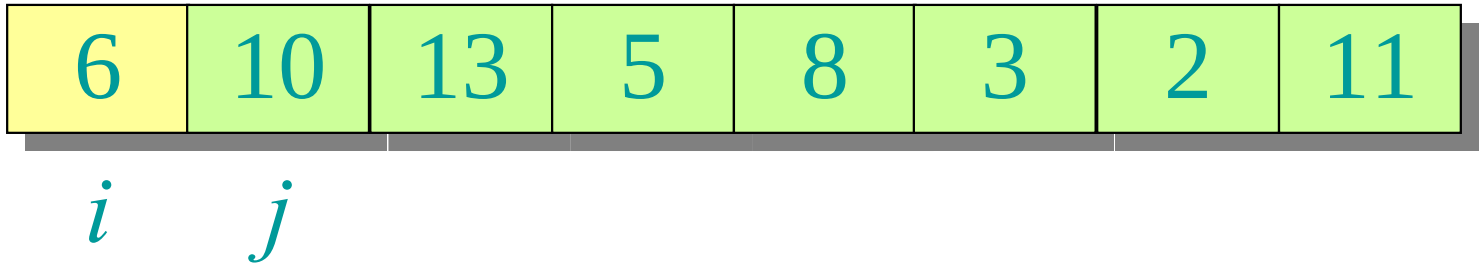
  2. j를 $l+1$에서 r까지 증가시키면서 아래 과정을 수행

     1. $t[j] < x$이면, i를 증가하고 i번째 수와 j번째 수를 교환

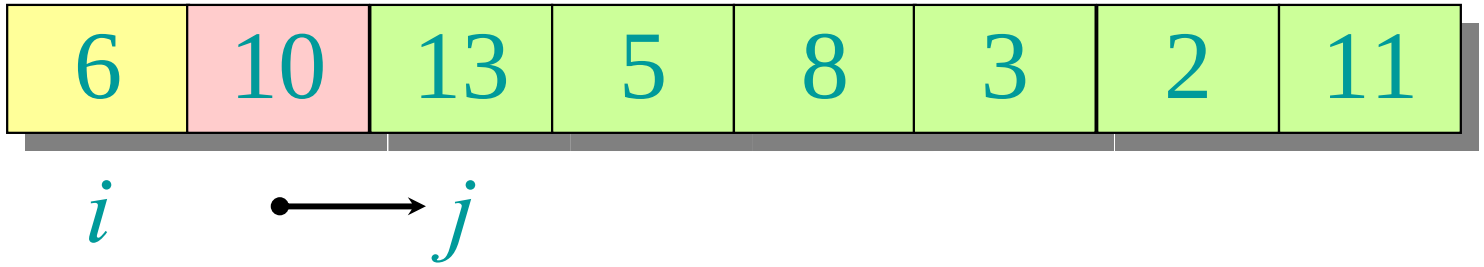
  3. i번째 수와 l번째 수(pivot value)를 교환



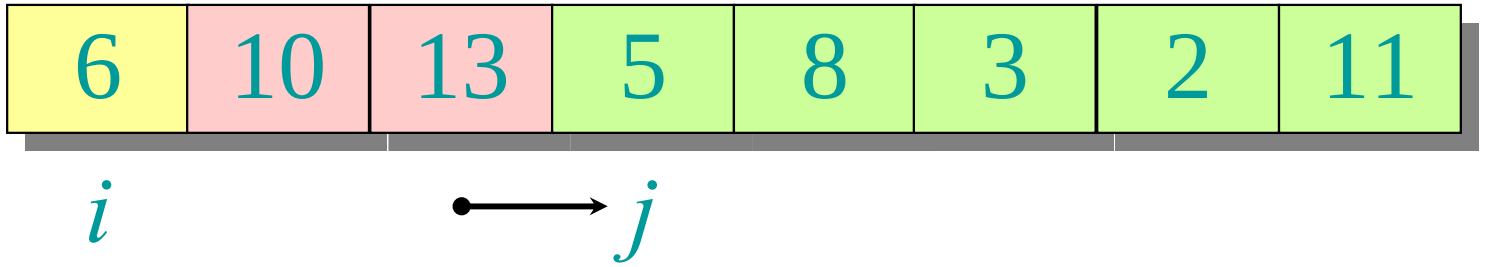
예시: 파티션 방법 2 (1/11)



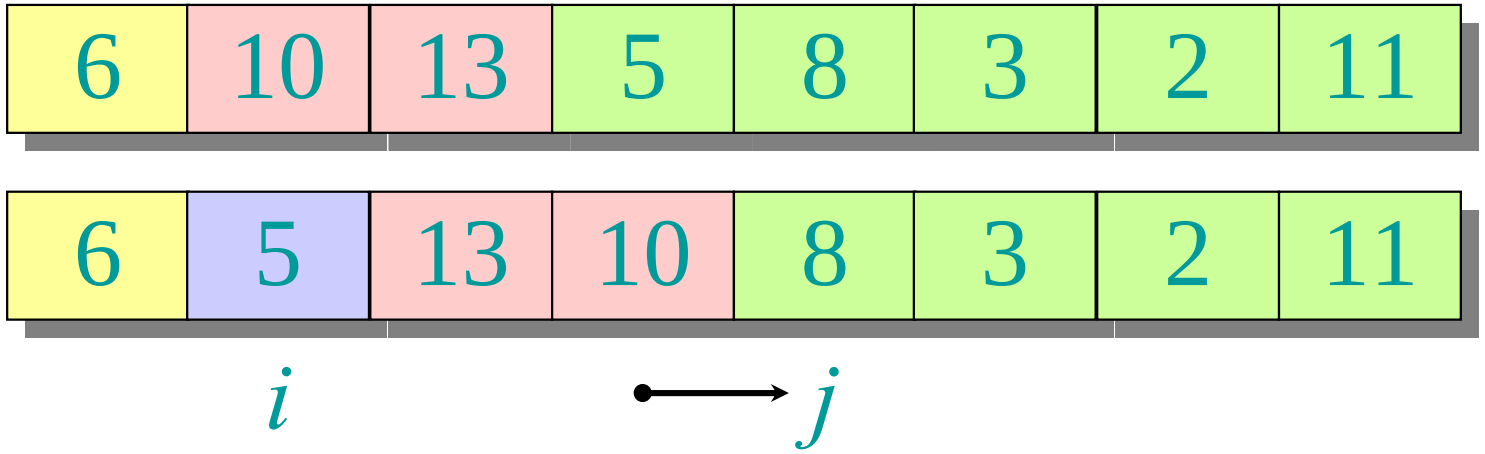
예시: 파티션 방법 2 (2/11)



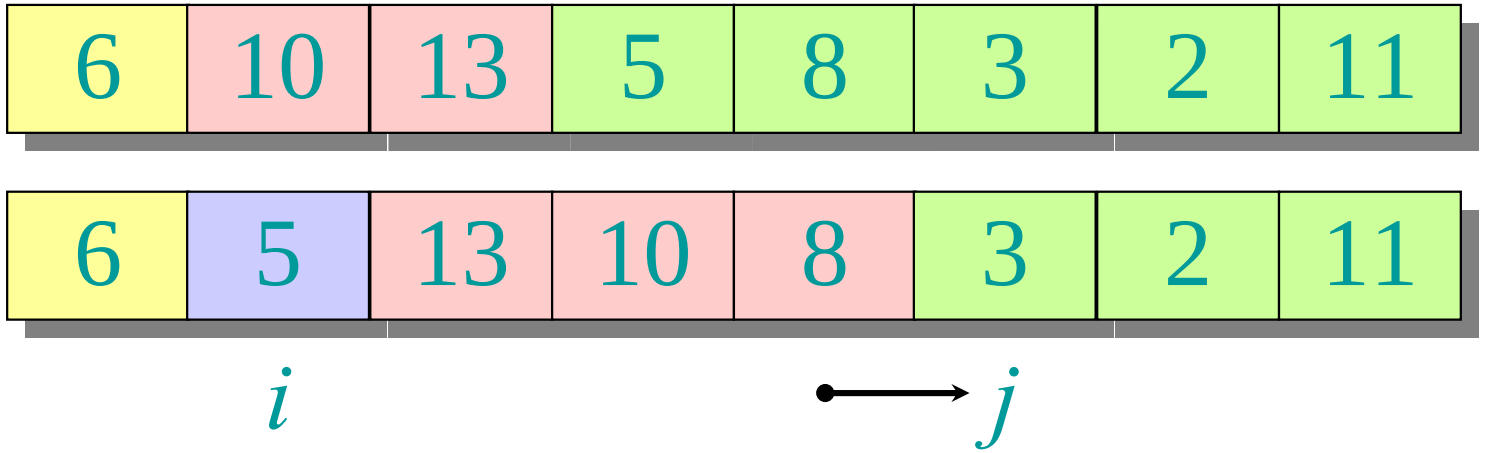
예시: 파티션 방법 2 (3/11)



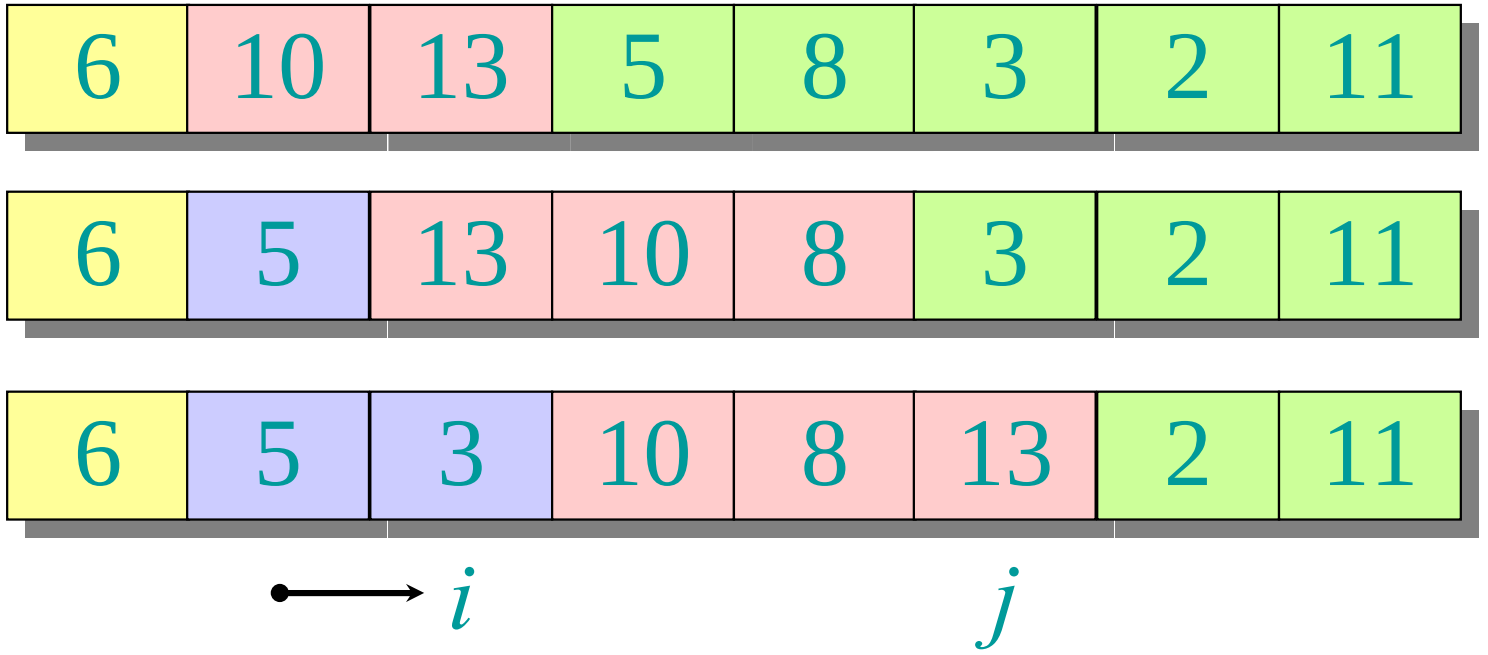
예시: 파티션 방법 2 (4/11)



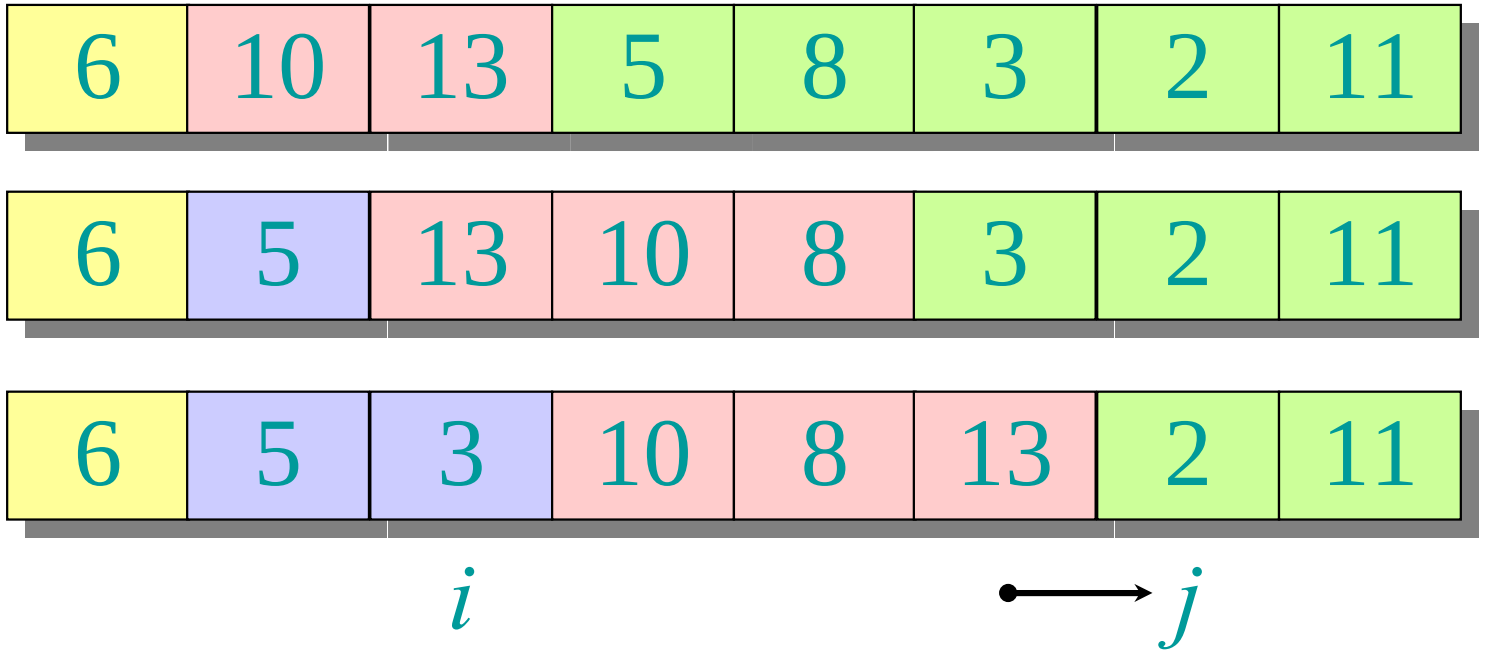
예시: 파티션 방법 2 (5/11)



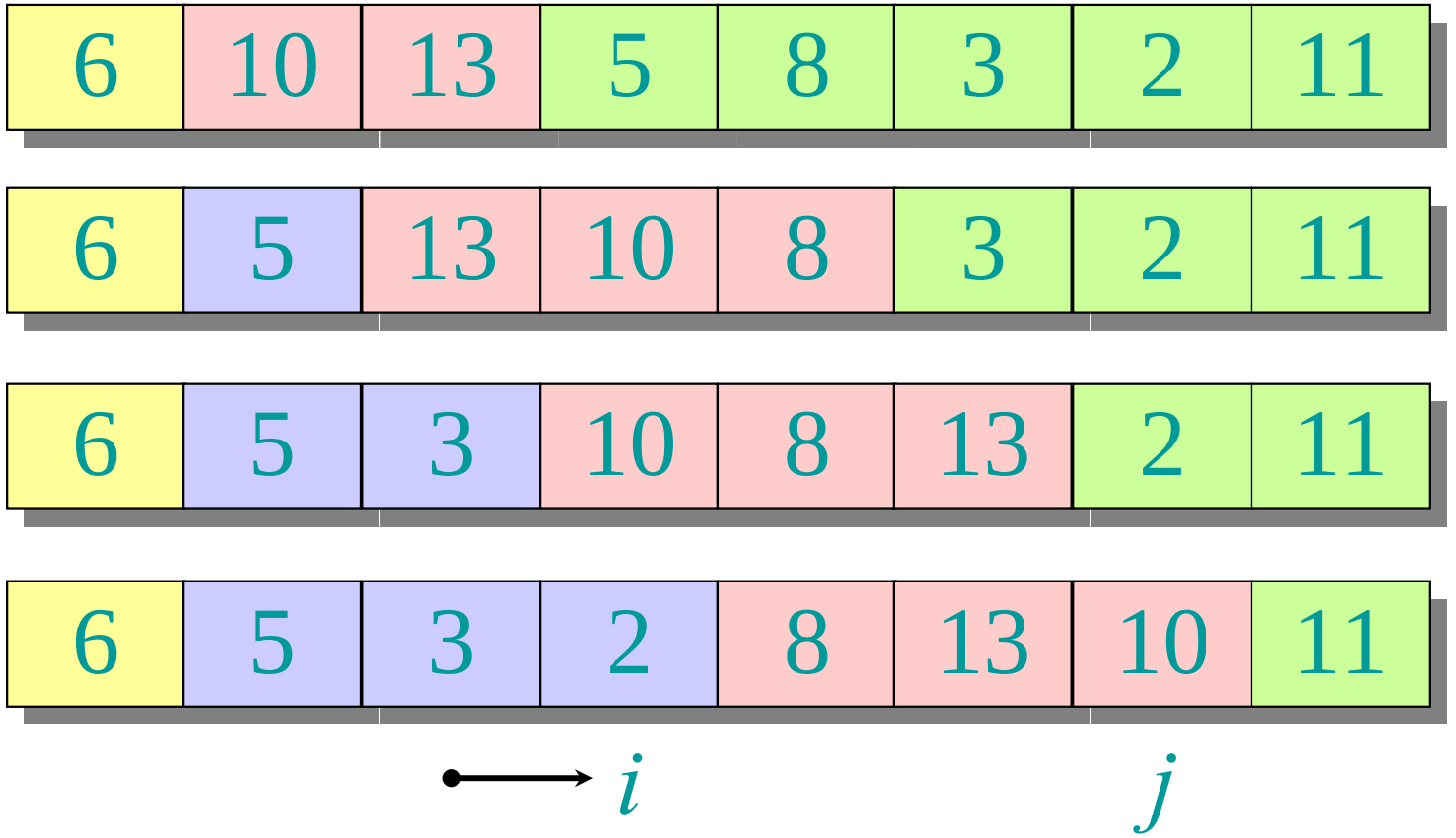
예시: 파티션 방법 2 (6/11)



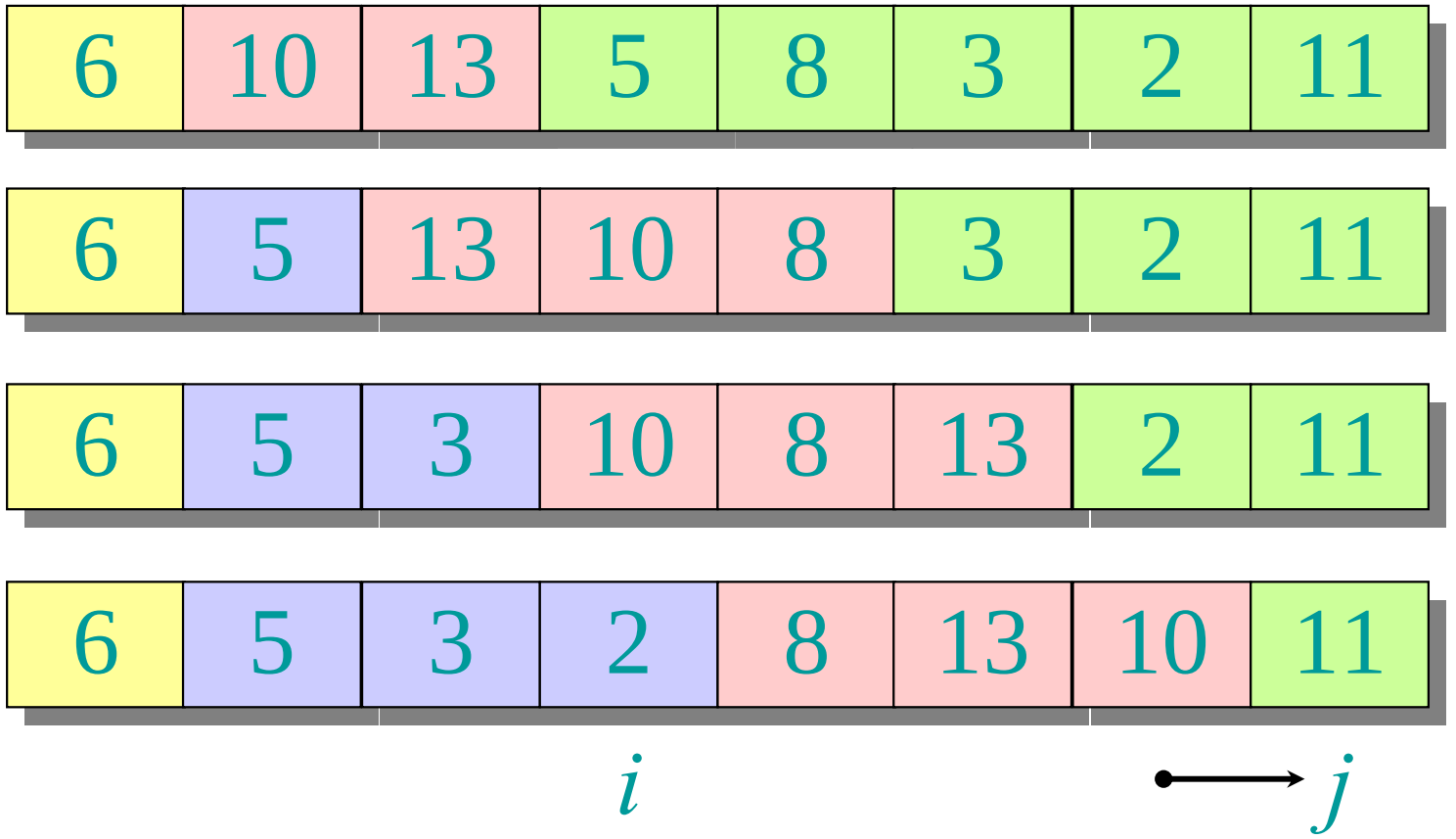
예시: 파티션 방법 2 (7/11)



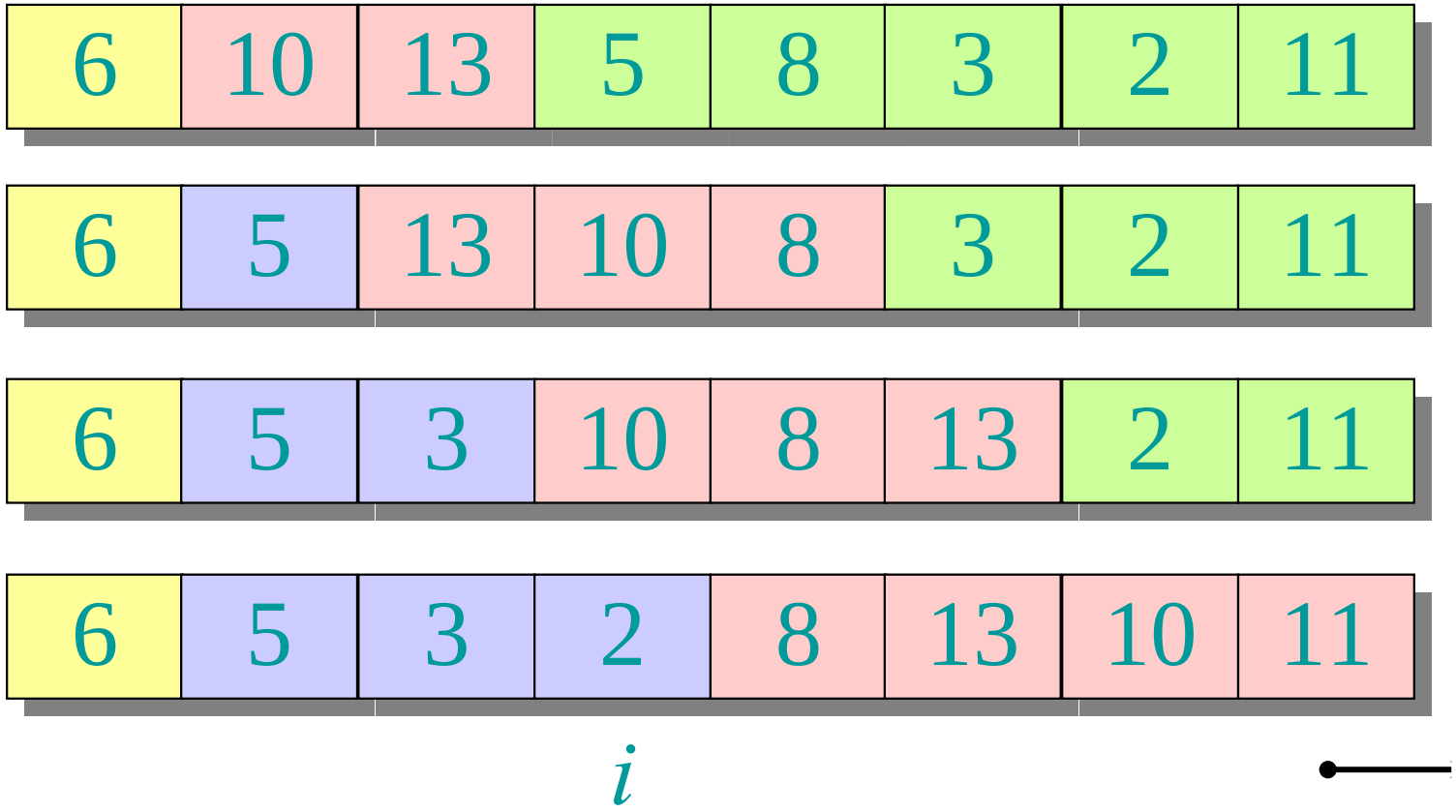
예시: 파티션 방법 2 (8/11)



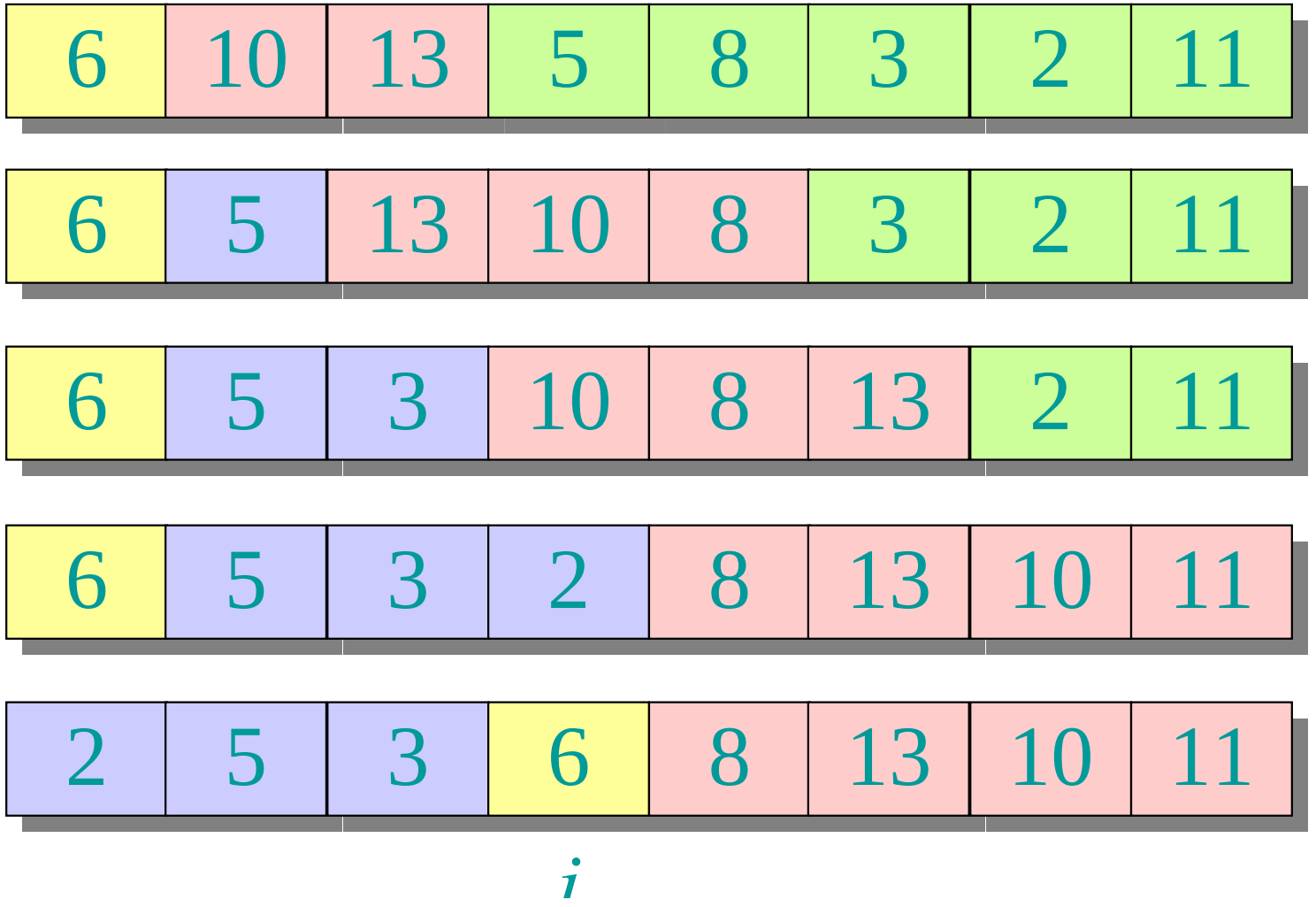
예시: 파티션 방법 2 (9/11)



예시: 파티션 방법 2 (10/11)



예시: 파티션 방법 2 (11/11)



Partition 구현

```python
def partition(t, l, r):  # method 1
    pivot = t[l]
    lm = l + 1
    rm = r

    while lm <= rm:
        while lm <= rm and t[lm] <= pivot:
            lm += 1

        while rm >= lm and t[rm] >= pivot:
            rm -= 1

        if lm < rm:
            t[lm], t[rm] = t[rm], t[lm]

    t[l], t[rm] = t[rm], t[l]
    return rm
```

```python
def partition(t, l, r):  # method 2
    pivot = t[l]
    i = l
    for j in range(l + 1, r + 1):
        if t[j] < pivot:
            i += 1
            t[i], t[j] = t[j], t[i]

    t[l], t[i] = t[i], t[l]

    return i
```

merge sort v.s. quick sort (1/2)

```
def merge_sort(t, l, r):
    # base case
    if l >= r:
        return

    # recursive case
    mid = (l + r) // 2
    merge_sort(t, l, mid)
    merge_sort(t, mid + 1, r)
    merge(t, l, r)
```

```
def quick_sort(t, l, r):
    # base case
    if l >= r:
        return

    # recursive case
    # p: location(index) of
pivot value
    p = partition(t, l, r)
    quick_sort(t, l, p - 1)
    quick_sort(t, p + 1, r)
```

merge sort v.s. quick sort (2/2)

- 비슷한 점: 리스트를 분할하여 작은 문제를 재귀적으로 정렬
- 다른 점: 분할과 결합 중, 어떤 부분이 복잡한지 (즉, 실제 정렬이 일어나게 되는지)

	merge sort	quick sort
분할	리스트를 (거의) 같은 크기의 리스트 둘로 나눔 → trivial	리스트를 pivot value보다 큰 수와 작은 수로 나눔 (partition) → complicated
정복	재귀적으로 정렬을 반복 base case: 리스트의 원소가 0 혹은 1개인 경우	
결합	정렬된 두 리스트를 하나의 리스트로 결합 (merge) → complicated	필요 없음 (분할과정에서 이미 큰 수와 작은 수로 나눔) → trivial

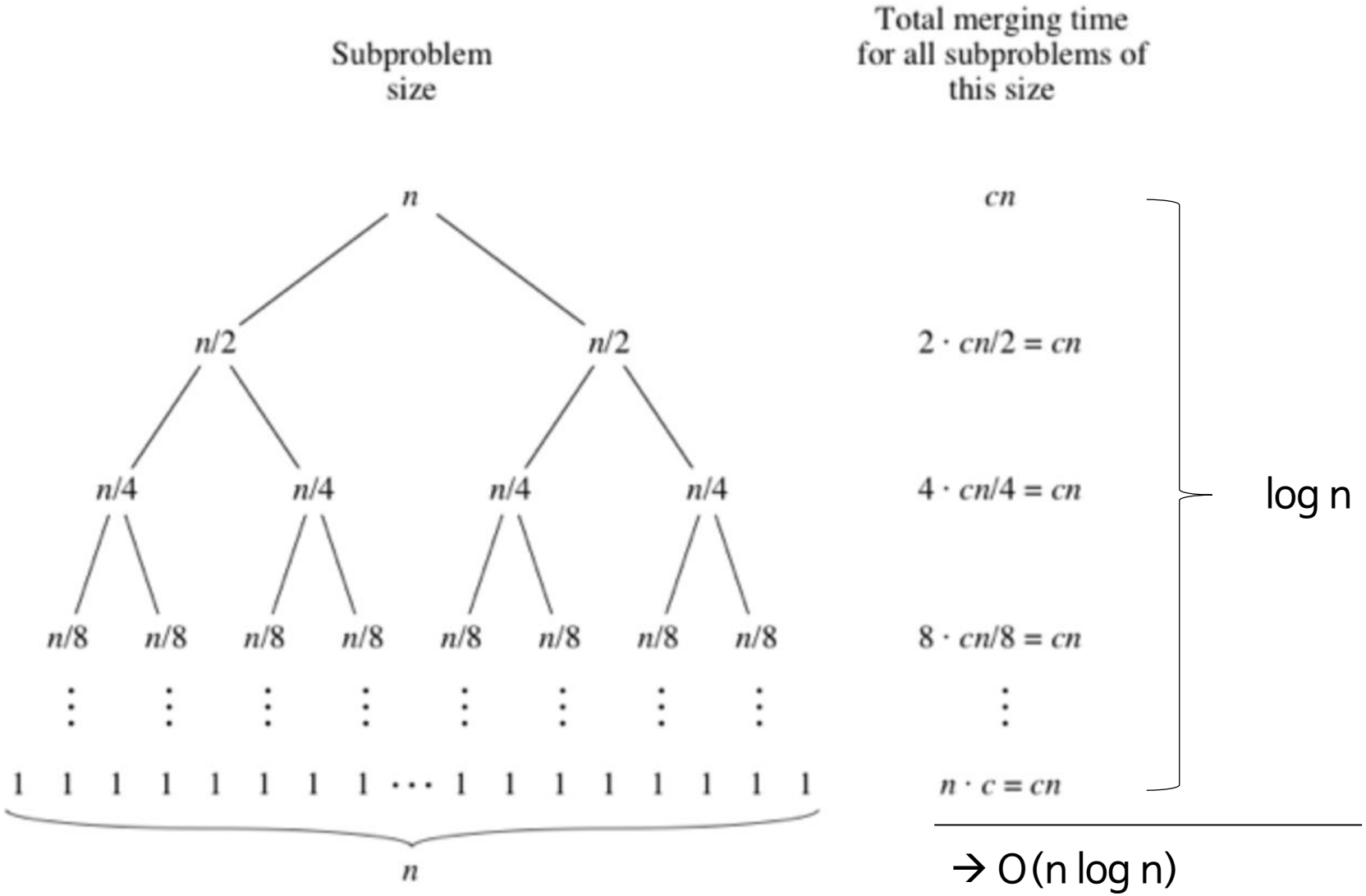
병합 정렬 복잡도 분석: merge_sort()

- merge_sort()함수에서,
  - 2개의 작은 문제(절반 정도의 크기를 가지는 문제)에 대해서 merge_sort()를 호출
  - merge() 함수를 호출
- merge_sort() 내부에서는 아주 간단한 작업만 함
- merge_sort() 함수가 merge() 함수를 호출했을 때 얼마만큼의 시간/공간이 필요한 지 분석

병합 정렬 복잡도 분석: merge()

- 두 개의 리스트를 합쳐서, n개의 아이템을 가지는 리스트를 만드는 경우
- 시간 복잡도
  - 두 개의 리스트에 있는 숫자를 비교해서 그 중 작은 숫자를 결과리스트(result)에 저장해야 함
  - 이 작업을 n번 반복해야 함 → $O(n)$
- 공간 복잡도
  - 중간결과 리스트를 저장할 추가적인 공간이 필요 → $O(n)$

병합 정렬 복잡도 분석: merge_sort() & merge()



알고리즘 복잡도 비교: selection, merge and quick sort

	Selection sort	Merge sort	Quick sort
시간 복잡도	$O(n^2)$	$O(n \log n)$	Average: $O(n \log n)$ Worst: $O(n^2)$
공간 복잡도	$O(1)$	$O(n)$	$O(1)$

읽을 거리

- 정렬 알고리즘
  - https://ko.wikipedia.org/wiki/정렬_알고리즘
  - https://namu.wiki/w/정렬%20알고리즘
- 알고리즘 실행 예시
  - 애니메이션: http://www.sorting-algorithms.com
  - 시각화: http://sortvis.org/visualisations.html
  - 청각화: http://flowingdata.com/2010/09/01/what-different-sorting-algorithms-sound-like/
  - 댄스화 (AlgoRhythmics): https://www.youtube.com/channel/UCIqiLefbVHsOAXDAxQJH7Xw
- 알고리즘 복잡도 분석
  - https://en.wikipedia.org/wiki/Analysis_of_algorithms

ANY QUESTIONS?