데이터 구조 입문

김종현

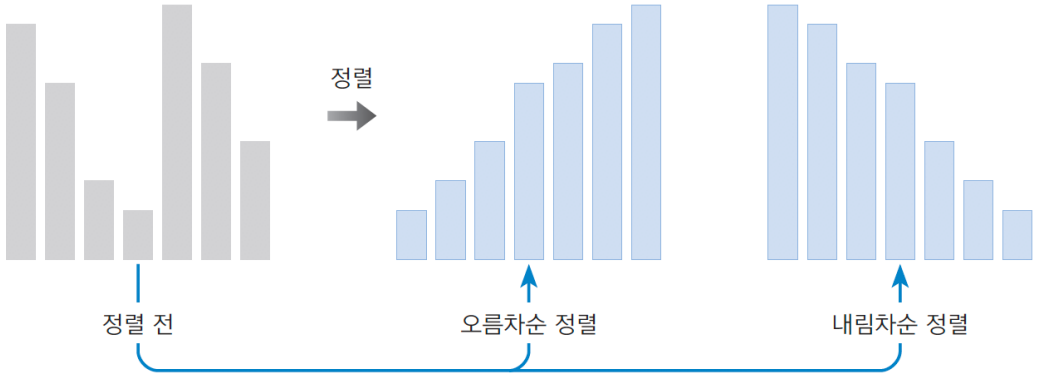
06

정렬 알고리즘

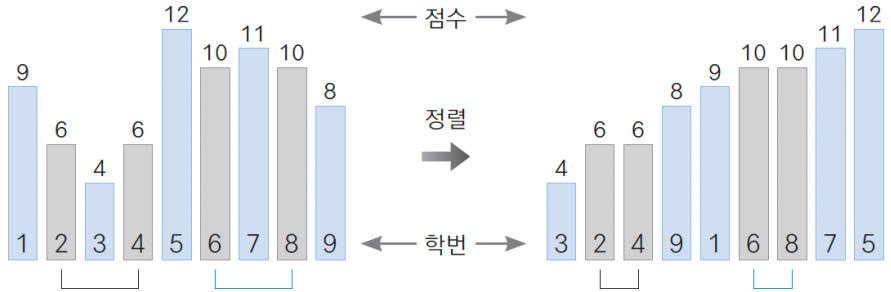
06-1 정렬 알고리즘

정렬이란?

- **정렬**sorting
  - 이름, 학번, 학점 등의 키key를 항목값의 대소 관계에 따라 데이터 집합을 일정한 순서로 바꾸어 늘어놓는 작업
  - **오름차순**ascending order
    - 값이 작은 데이터를 앞쪽에 늘어놓는 것
  - **내림차순**descending order
    - 값이 큰 데이터를 앞쪽에 늘어놓는 것



- **안정적인**stable **정렬 알고리즘**
  - 값이 같은 원소의 순서가 정렬한 후에도 유지되는 것



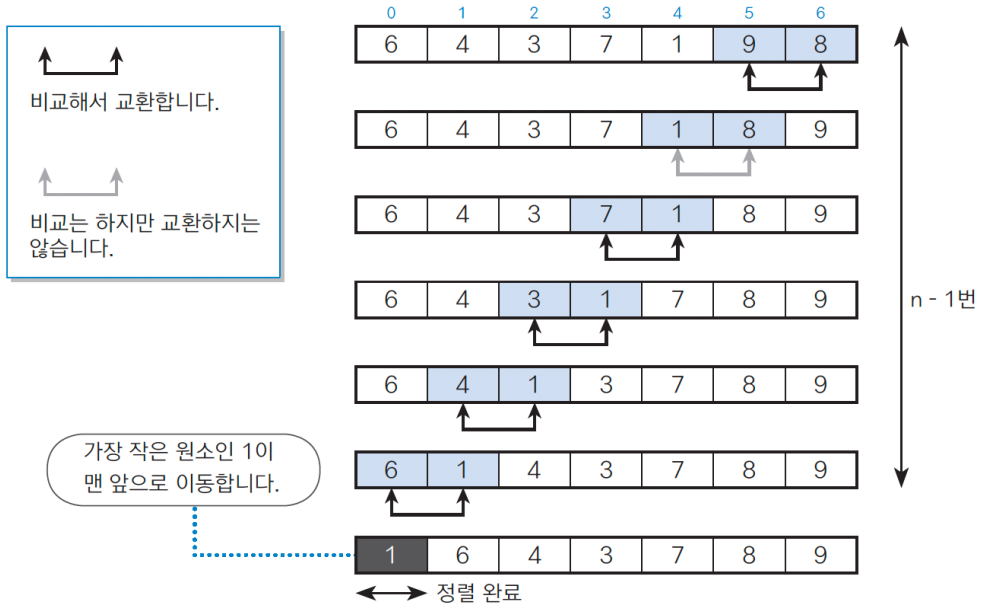
값이 같은 원소의 순서는 정렬한 후에도 같습니다.

- **내부 정렬**internal sorting
  - 정렬할 모든 데이터를 하나의 배열에 저장할 수 있는 경우 사용하는 알고리즘
- **외부 정렬**external sorting
  - 정렬할 데이터가 많아서 하나의 배열에서 저장할 수 없는 경우 사용하는 알고리즘

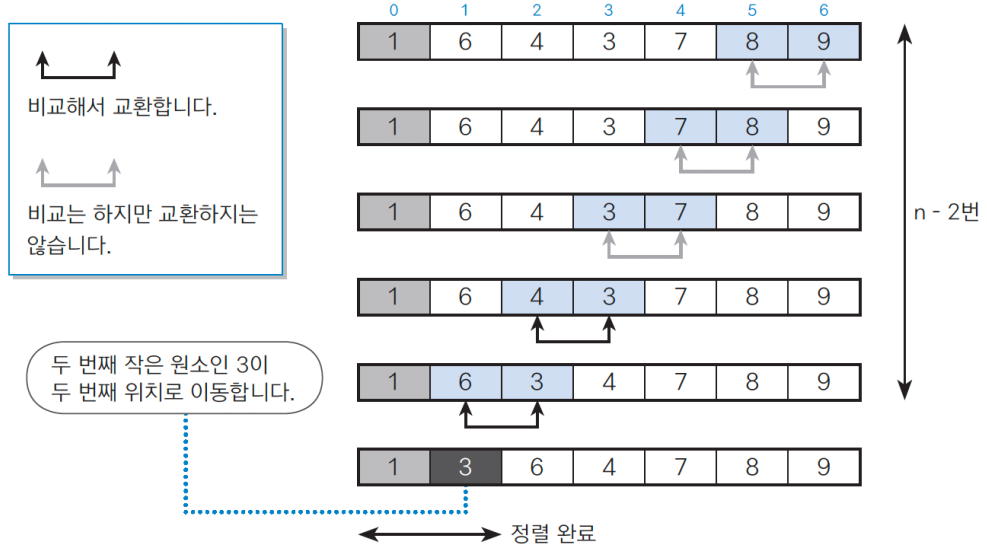
06-2 버블 정렬

버블 정렬 알아보기 – (1)

- **버블 정렬**bubble sort
  - 이웃한 두 원소의 대소 관계를 비교하여 필요에 따라 교환을 반복하는 알고리즘
  - 단순 교환 정렬이라고도 함



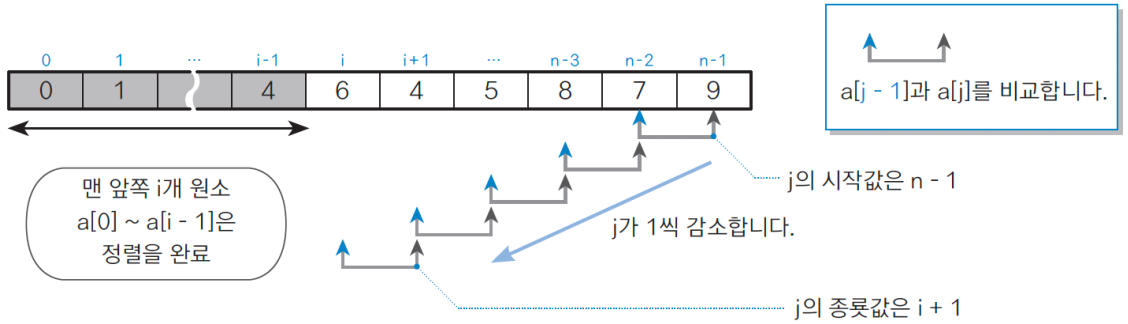
- **패스**pass
  - 비교 · 교환하는 과정
- 모든 정렬이 끝나려면 패스를 n – 1번 수행



06-2 버블 정렬

버블 정렬 알아보기 – (2)

- **실습 6-1**
  - 버블 정렬 알고리즘
  - 두 원소 a[j – 1]과 a[j]의 값을 비교하여 앞쪽 값이 뒷쪽 값보다 크면 교환



  - 원소 비교 횟수

$$(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$$

    - 실제 원소를 교환하는 횟수는 배열의 원솟값에 영향을 받으므로 평균값은 절반인 n(n – 1) / 4번

Do it! 실습 6-1 • 완성 파일 chap06/bubble_sort1.py

```
01: # 버블 정렬 알고리즘 구현하기
02:
03: from typing import MutableSequence
04:
05: def bubble_sort(a: MutableSequence) -> None:
06:     """버블 정렬"""
07:     n = len(a)
08:     for i in range(n - 1):
09:         for j in range(n - 1, i, -1):
10:             if a[j - 1] > a[j]:
11:                 a[j - 1], a[j] = a[j], a[j - 1]
12:
13: if __name__ == '__main__':
14:     print('버블 정렬을 수행합니다.')
15:     num = int(input('원소 수를 입력하세요.: '))
16:     x = [None] * num  # 원소 수가 num인 배열을 생성
17:
18:     for i in range(num):
19:         x[i] = int(input(f'x[{i}]: '))
20:
21:     bubble_sort(x)    # 배열 x를 버블 정렬
22:
23:     print('오름차순으로 정렬했습니다.')
24:     for i in range(num):
25:         print(f'x[{i}] = {x[i]}')
```

패스

▶ 실행 결과

```
버블 정렬을 수행합니다.
원소 수를 입력하세요.: 7
x[0]: 6
x[1]: 4
x[2]: 3
x[3]: 7
x[4]: 1
x[5]: 9
x[6]: 8
오름차순으로 정렬했습니다.
x[0] = 1
x[1] = 3
x[2] = 4
x[3] = 6
x[4] = 7
x[5] = 8
x[6] = 9
```

06-2 버블 정렬

버블 정렬 알아보기 – (3)

- **실습 6-2**
  - 버블 정렬 과정을 상세하게 출력하도록 수정한 프로그램
  - 비교하는 두 원소 사이에 교환할 경우 +, 교환하지 않을 경우 – 출력

Do it! 실습 6-2 • 완성 파일 chap06/bubble_sort1_verbose.py

```
01: # 버블 정렬 알고리즘 구현하기(정렬 과정을 출력)
02:
03: from typing import MutableSequence
04:
05: def bubble_sort_verbose(a: MutableSequence) -> None:
06:     """버블 정렬(정렬 과정을 출력)"""
07:     ccnt = 0  # 비교 횟수
08:     scnt = 0  # 교환 횟수
09:     n = len(a)
10:     for i in range(n - 1):
11:         print(f'패스 {i + 1}')
12:         for j in range(n - 1, i, -1):
13:             for m in range(0, n - 1):
14:                 print(f'{a[m]:2}' + ('  ' if m != j - 1 else
15:                                       ' +' if a[j - 1] > a[j] else ' -'),
16:                                       end='')
17:             print(f'{a[n - 1]:2}')
18:             ccnt += 1
19:             if a[j - 1] > a[j]:
20:                 scnt += 1
21:                 a[j - 1], a[j] = a[j], a[j - 1]
22:         for m in range(0, n - 1):
23:             print(f'{a[m]:2}', end='  ')
24:         print(f'{a[n - 1]:2}')
25:     print(f'비교를 {ccnt}번 했습니다.')
26:     print(f'교환을 {scnt}번 했습니다.')
(… 생략 …)
```

▶ 실행 결과

```
버블 정렬을 수행합니다.
원소 수를 입력하세요.: 7
(… 생략 …)
패스 1
 6  4  3  7  1  9 +8
 6  4  3  7  1 -8  9
 6  4  3  7 +1  8  9
 6  4  3 +1  7  8  9
 6  4 +1  3  7  8  9
 6 +1  4  3  7  8  9
 1  6  4  3  7  8  9
패스 2
 1  6  4  3  7  8 -9
 1  6  4  3  7 -8  9
 1  6  4  3 -7  8  9
 1  6  4 +3  7  8  9
 1  6 +3  4  7  8  9
 1  3  6  4  7  8  9
패스 3
 1  3  6  4  7  8 -9
 1  3  6  4  7 -8  9
 1  3  6  4 -7  8  9
 1  3  6 +4  7  8  9
 1  3  4  6  7  8  9
패스 4
 1  3  4  6  7  8 -9
 1  3  4  6  7 -8  9
 1  3  4  6 -7  8  9
 1  3  4  6  7  8  9
패스 5
 1  3  4  6  7  8 -9
 1  3  4  6  7 -8  9
 1  3  4  6  7  8  9
패스 6
 1  3  4  6  7  8 -9
 1  3  4  6  7  8  9
비교를 21번 했습니다.
교환을 8번 했습니다.
오름차순으로 정렬했습니다.
(… 생략 …)
```

06-2 버블 정렬

버블 정렬 알아보기 – (4)

- **실습 6-3**
  - 교환 횟수에 따라 중단 방식을 적용하여 개선한 프로그램
  - 어떤 패스의 원소 교환 횟수가 0이면 모든 원소가 정렬을 완료한 경우이므로 그 이후의 패스는 불필요하다고 판단하여 정렬을 중단
  - 이러한 중단 방식을 적용하면 정렬을 모두 마쳤거나 정렬이 거의 다 된 배열에서 비교 연산이 크게 줄어들어 실행 시간 단축 가능
  - 실습 6-2와 비교 횟수 차이: **21번 → 18번**

Do it! 실습 6-3

• 완성 파일 chap06/bubble_sort2.py

```
01: # 버블 정렬 알고리즘 구현하기(알고리즘의 개선 1)
02:
03: from typing import MutableSequence
04:
05: def bubble_sort(a: MutableSequence) -> None:
06:     """버블 정렬(교환 횟수에 따른 중단)"""
07:     n = len(a)
08:     for i in range(n - 1):
09:         exchng = 0  # 패스에서 교환 횟수
10:         for j in range(n - 1, i, -1):
11:             if a[j - 1] > a[j]:
12:                 a[j - 1], a[j] = a[j], a[j - 1]
13:                 exchng += 1
14:         if exchng == 0:
15:             break
(… 생략 …)
```

패스

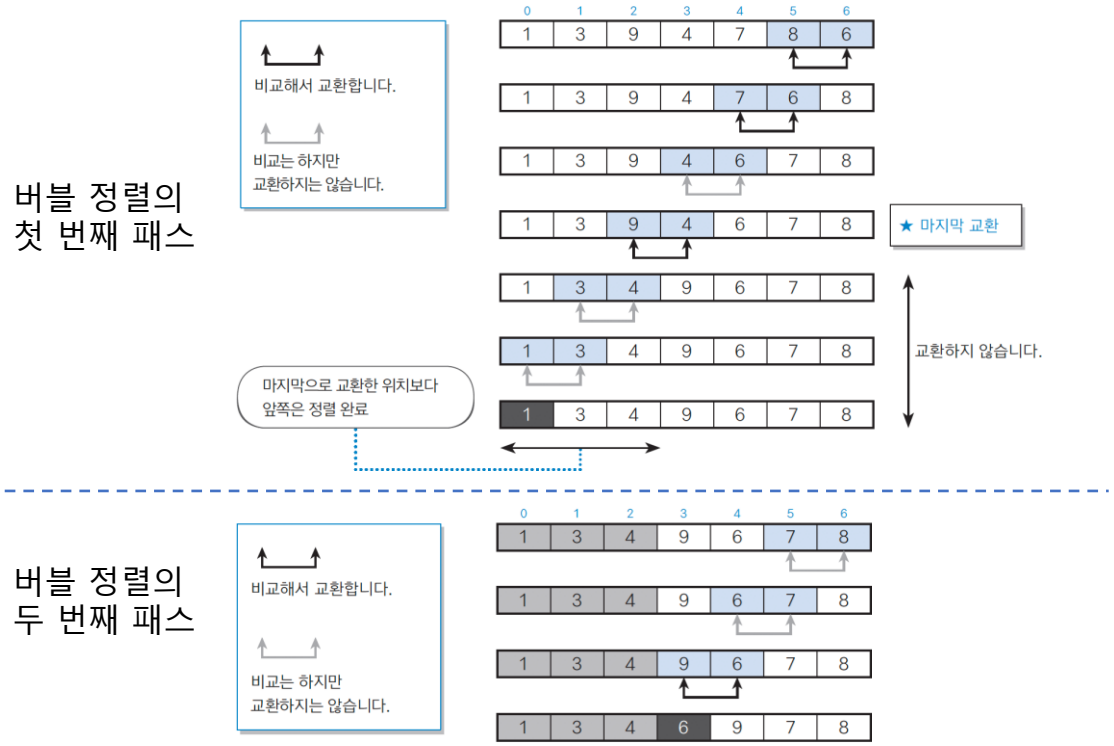
▶ 실행 결과

```
(… 생략 …)
패스 1
 6  4  3  7  1  9+8
 6  4  3  7  1-8  9
 6  4  3  7+1  8  9
 6  4  3+1  7  8  9
 6  4+1  3  7  8  9
 6+1  4  3  7  8  9
 1  6  4  3  7  8  9
패스 2
 1  6  4  3  7  8-9
 1  6  4  3  7-8  9
 1  6  4  3-7  8  9
 1  6  4+3  7  8  9
 1  6+3  4  7  8  9
 1  3  6  4  7  8  9
패스 3
 1  3  6  4  7  8-9
 1  3  6  4  7-8  9
 1  3  6  4-7  8  9
 1  3  6+4  7  8  9
 1  3  4  6  7  8  9
패스 4
 1  3  4  6  7  8-9
 1  3  4  6  7-8  9
 1  3  4  6-7  8  9
 1  3  4  6  7  8  9
비교를 18번 했습니다.
교환을 8번 했습니다.
(… 생략 …)
```

06-2 버블 정렬

버블 정렬 알아보기 – (5)

- **실습 6-4**
  - 이미 정렬된 원소를 제외한 나머지만 비교 · 교환하도록 스캔 범위를 제한하는 방법으로 개선한 프로그램



Do it! 실습 6-4 • 완성 파일 chap06/bubble_sort3.py

```
01: # 버블 정렬 알고리즘 구현하기(알고리즘의 개선 2)
02:
03: from typing import MutableSequence
04:
05: def bubble_sort(a: MutableSequence) -> None:
06:     """버블 정렬(스캔 범위를 제한)"""
07:     n = len(a)
08:     k = 0
09:     while k < n - 1:
10:         last = n - 1
11:         for j in range(n - 1, k, -1):
12:             if a[j - 1] > a[j]:
13:                 a[j - 1], a[j] = a[j], a[j - 1]
14:                 last = j
15:         k = last
```

패스

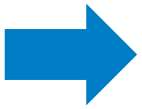
(… 생략 …)

06-2 버블 정렬

버블 정렬 알아보기 – (6)

- **실행 결과 비교하기**

1	3	9	4	7	8	6



정렬

Do it! 실습 6-2 실행 결과

(… 생략 …)
패스 1
1 3 9 4 7 8+6
1 3 9 4 7+6 8
1 3 9 4-6 7 8
1 3 9+4 6 7 8
1 3-4 9 6 7 8
1-3 4 9 6 7 8
1 3 4 9 6 7 8
패스 2
1 3 4 9 6 7-8
1 3 4 9 6-7 8
1 3 4 9+6 7 8
1 3 4-6 9 7 8
1 3-4 6 9 7 8
1 3 4 6 9 7 8
패스 3
1 3 4 6 9 7-8
1 3 4 6 9+7 8
1 3 4 6-7 9 8
1 3 4-6 7 9 8
1 3 4 6 7 9 8
(… 생략 … 패스 6까지 수행)
비교를 21번 했습니다.
교환을 6번 했습니다.

Do it! 실습 6-3 실행 결과

(… 생략 …)
패스 1
1 3 9 4 7 8+6
1 3 9 4 7+6 8
1 3 9 4-6 7 8
1 3 9+4 6 7 8
1 3-4 9 6 7 8
1-3 4 9 6 7 8
1 3 4 9 6 7 8
패스 2
1 3 4 9 6 7-8
1 3 4 9 6-7 8
1 3 4 9+6 7 8
1 3 4-6 9 7 8
1 3-4 6 9 7 8
1 3 4 6 9 7 8
패스 3
1 3 4 6 9 7-8
1 3 4 6 9+7 8
1 3 4 6-7 9 8
1 3 4-6 7 9 8
1 3 4 6 7 9 8
(… 생략 … 패스 5까지 수행)
비교를 20번 했습니다.
교환을 6번 했습니다.

Do it! 실습 6-4 실행 결과

(… 생략 …)
패스 1
1 3 9 4 7 8+6
1 3 9 4 7+6 8
1 3 9 4-6 7 8
1 3 9+4 6 7 8
1 3-4 9 6 7 8
1-3 4 9 6 7 8
1 3 4 9 6 7 8
패스 2
1 3 4 9 6 7-8
1 3 4 9 6-7 8
1 3 4 9+6 7 8
1 3 4 6 9 7 8
패스 3
1 3 4 6 9 7-8
1 3 4 6 9+7 8
1 3 4 6 7 9 8
패스 4
1 3 4 6 7 9+8
1 3 4 6 7 8 9
비교를 12번 했습니다.
교환을 6번 했습니다.

06-2 버블 정렬

셰이커 정렬 알아보기 – (1)

- **버블 정렬 프로그램의 한계**
  - 정렬이 거의 완료된 아래 배열을 버블 정렬 프로그램으로 정렬하면?

9	1	3	4	6	7	8

  - 가장 큰 원소인 9가 맨 앞에 있고, 한 패스에 하나씩 뒤로 이동하기 때문에 정렬 작업을 빠르게 마칠 수 없음

▶ **Do it!** 실습 6-2, 6-3, 6-4 실행 결과

(… 생략 …)

패스 1
9 1 3 4 6 7 - 8
9 1 3 4 6 - 7 8
9 1 3 4 - 6 7 8
9 1 3 - 4 6 7 8
9 1 - 3 4 6 7 8
9 + 1 3 4 6 7 8
1 9 3 4 6 7 8

패스 2
1 9 3 4 6 7 - 8
1 9 3 4 6 - 7 8
1 9 3 4 - 6 7 8
1 9 3 - 4 6 7 8
1 9 + 3 4 6 7 8
1 3 9 4 6 7 8

패스 3
1 3 9 4 6 7 - 8
1 3 9 4 6 - 7 8
1 3 9 4 - 6 7 8
1 3 9 + 4 6 7 8
1 3 4 9 6 7 8

패스 4
1 3 4 9 6 7 - 8
1 3 4 9 6 - 7 8
1 3 4 9 + 6 7 8
1 3 4 6 9 7 8

패스 5
1 3 4 6 9 7 - 8
1 3 4 6 9 + 7 8
1 3 4 6 7 9 8

패스 6
1 3 4 6 7 9 + 8
1 3 4 6 7 8 9

비교를 21번 했습니다.
교환을 6번 했습니다.

06-2 버블 정렬

셰이커 정렬 알아보기 – (2)

- **셰이커 정렬**shaker sort
  - 홀수 패스에서는 가장 작은 원소를 맨 앞으로 이동시키고, 짝수 패스에서는 가장 큰 원소를 맨 뒤로 이동시켜 패스의 스캔 방향을 번갈아 바꾸는 방법으로 버블 정렬을 개선한 알고리즘
  - 양방향 버블 정렬bidirectional bubble sort, 칵테일 정렬cocktail sort, 칵테일 셰이커 정렬cocktail shaker sort이라고 함

- **실습 6-5**
  - 실습 6-4의 버블 정렬을 셰이커 정렬로 개선하여 shaker_sort( ) 함수를 사용하는 프로그램
  - 실습 6-4와 비교 횟수 차이: **21번 → 10번**

Do it! 실습 6-5

• 완성 파일 chap06/shaker_sort.py

```
01: # 셰이커 정렬 알고리즘 구현하기
02:
03: from typing import MutableSequence
04:
05: def shaker_sort(a: MutableSequence) -> None:
06:     """셰이커 정렬"""
07:     left = 0
08:     right = len(a) - 1
09:     last = right
10:     while left < right:
11:         for j in range(right, left, -1):
12:             if a[j - 1] > a[j]:
13:                 a[j - 1], a[j] = a[j], a[j - 1]
14:                 last = j
15:         left = last
16:
17:         for j in range(left, right):
18:             if a[j] > a[j + 1]:
19:                 a[j], a[j + 1] = a[j + 1], a[j]
20:                 last = j
21:         right = last
(… 생략 …)
```

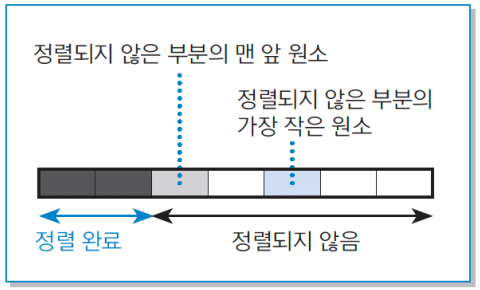
실행 결과

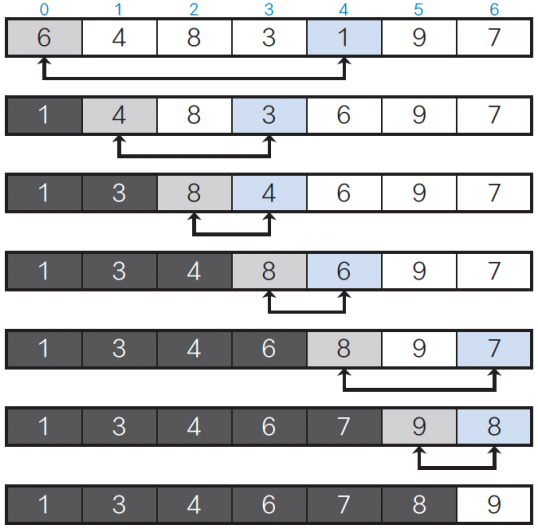
```
(… 생략 …)
패스 1
  9  1  3  4  6  7- 8
  9  1  3  4  6- 7  8
  9  1  3  4- 6  7  8
  9  1  3- 4  6  7  8
  9  1- 3  4  6  7  8
  9+ 1  3  4  6  7  8
  1  9  3  4  6  7  8
패스 2
  1  9+ 3  4  6  7  8
  1  3  9+ 4  6  7  8
  1  3  4  9+ 6  7  8
  1  3  4  6  9+ 7  8
  1  3  4  6  7  9+ 8
  1  3  4  6  7  8  9
패스 3
  1  3  4  6  7- 8  9
  1  3  4  6- 7  8  9
  1  3  4- 6  7  8  9
  1  3- 4  6  7  8  9
  1  3  4  6  7  8  9
비교를 10번 했습니다.
교환을 6번 했습니다.
(… 생략 …)
```

06-3 단순 선택 정렬

단순 선택 정렬 알아보기 – (1)

- **단순 선택 정렬**straight selection sort
  - 가장 작은 원소부터 선택해 알맞은 위치로 옮기는 작업을 반복하며 정렬하는 알고리즘





- 단순 선택 정렬에서 교환 과정

1. 아직 정렬하지 않은 부분에서 값이 가장 작은 원소 a[min] 선택
2. a[min]과 아직 정렬하지 않은 부분에서 맨 앞에 있는 원소 교환

- 이 과정을 n – 1번 반복하면 정렬하지 않은 부분이 없어지면서 전체 정렬 완료

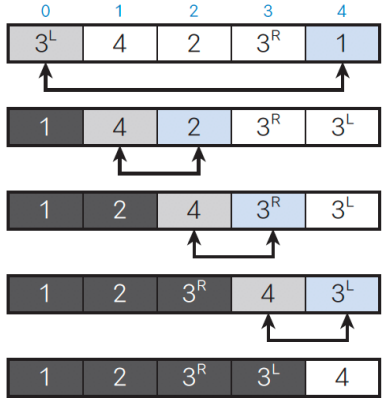
- 알고리즘의 개요

```
for i in range(n - 1):
    min   # a[i], …, a[n - 1]에서 키값이 가장 작은 원소의 인덱스
    a[i]와 a[min]의 값을 교환합니다.
```

06-3 단순 선택 정렬

단순 선택 정렬 알아보기 – (2)

- **실습 6-6**
  - 단순 선택 정렬 함수 selection_sort( )를 수행하는 프로그램
  - 단순 선택 정렬 알고리즘의 원솟값을 비교하는 횟수 $(n^2 - n) / 2$번
  - 서로 이웃하지 않는 떨어져 있는 원소를 교환하므로 안정적이지 않은 정렬 알고리즘
    - 원래 앞에 있던 3을 3^L, 뒤에 있던 3을 3^R이라고 표기
    - 결국 두 원소의 순서는 정렬한 후 뒤바뀜



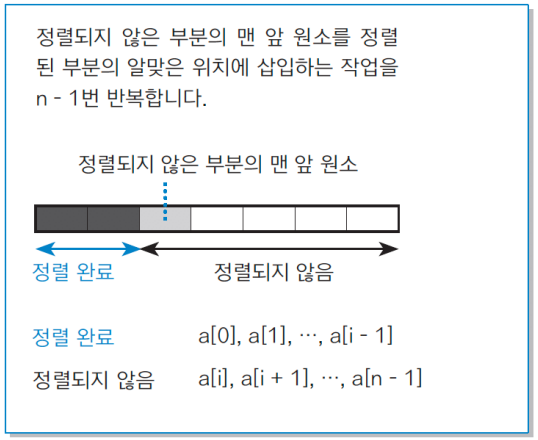
Do it! 실습 6-6 • 완성 파일 chap06/selection_sort.py

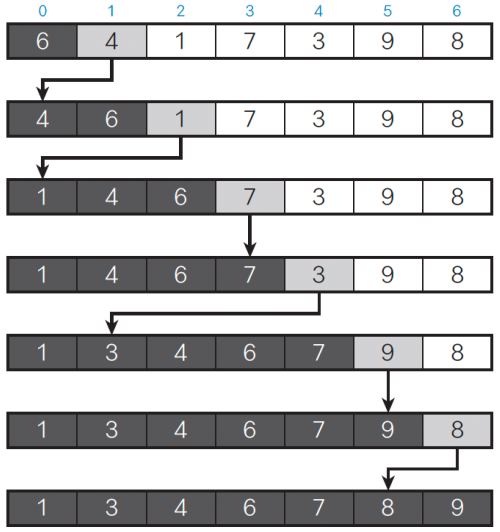
```
01: # 단순 선택 정렬 알고리즘 구현하기
02:
03: from typing import MutableSequence
04:
05: def selection_sort(a: MutableSequence) -> None:
06:     """단순 선택 정렬"""
07:     n = len(a)
08:     for i in range(n - 1):
09:         min = i   # 정렬할 부분에서 가장 작은 원소의 인덱스
10:         for j in range(i + 1, n):
11:             if a[j] < a[min]:
12:                 min = j
13:         a[i], a[min] = a[min], a[i]  # 정렬할 부분에서 맨 앞의 원소와 가장 작은 원소를 교환
( …생략… )
```

06-4 단순 삽입 정렬

단순 삽입 정렬 알아보기 – (3)

- **단순 삽입 정렬**straight insertion sort
  - 주목한 원소보다 더 앞쪽에서 알맞은 위치로 삽입하며 정렬하는 알고리즘
  - 단순 선택 정렬과 비슷해 보이지만 값이 가장 작은 원소를 선택하지 않는다는 점이 다름





- 단순 삽입 정렬의 삽입 과정

> 아직 정렬되지 않은 부분의 맨 앞 원소를 정렬된 부분의 알맞은 위치에 삽입

- 이 과정을 n – 1번 반복하면 정렬 완료

- 알고리즘의 개요

```
for i in range(1, n):
    tmp ← a[i]를 넣습니다.
    tmp를 a[0], ···, a[i - 1]의 알맞은 위치에 삽입합니다.
```

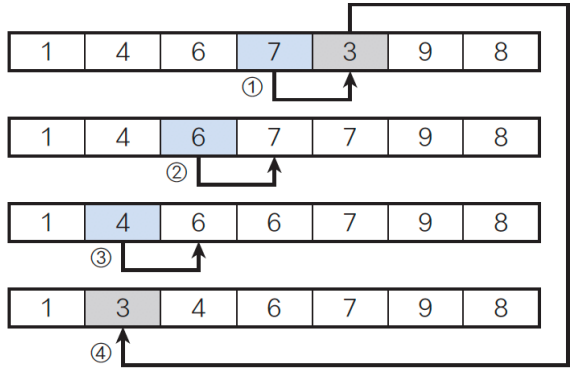
06-4 단순 삽입 정렬

단순 삽입 정렬 알아보기 – (4)

- **단순 삽입 정렬**straight insertion sort
  - 단순 삽입 정렬에서 원소 삽입

①~③ … 3보다 작은 원소를 만날 때까지 이웃한 왼쪽 원소를 하나씩 대입하는 작업을 반복합니다.
④ … 멈춘 위치에 3을 대입합니다.

```
j = i
tmp = a[i]
while j > 0 and a[j - 1] > tmp:
    a[j] = a[j - 1]
    j -= 1
a[j] = tmp
```



- 종료 조건

1. 정렬된 배열의 왼쪽 끝에 도달한 경우
2. tmp보다 작거나 키값이 같은 원소 a[j– 1]을 발견할 경우

- 스캔 작업 반복 조건

1. j가 0보다 큰 경우
2. a[j – 1]의 값이 tmp보다 큰 경우

  - 반복 제어 변수 j에 i를, tmp에 a[i]를 대입하고 종료 조건을 만족할 때까지 j를 1씩 감소시키면서 대입 작업 반복

06-4 단순 삽입 정렬

단순 삽입 정렬 알아보기 – (5)

- **실습 6-7**
  - 단순 삽입 정렬을 수행하는 프로그램
  - 이 알고리즘은 서로 떨어져 있는 원소를 교환하지 않으므로 안정적인 정렬 알고리즘
  - 원소의 비교 횟수와 교환 횟수는 모두 $n^2 / 2$번
- **단순 정렬 알고리즘의 시간 복잡도**
  - 단순 정렬(버블, 선택, 삽입) 알고리즘의 시간 복잡도는 모두 $O(n^2)$으로 프로그램의 효율이 좋지 않음

Do it! 실습 6-7 • 완성 파일 chap06/insertion_sort.py

```
01: # 단순 삽입 정렬 알고리즘 구현하기
02:
03: from typing import MutableSequence
04:
05: def insertion_sort(a: MutableSequence) -> None:
06:     """단순 삽입 정렬"""
07:     n = len(a)
08:     for i in range(1, n):
09:         j = i
10:         tmp = a[i]
11:         while j > 0 and a[j - 1] > tmp:
12:             a[j] = a[j - 1]
13:             j -= 1
14:         a[j] = tmp
15:
16: if __name__ == '__main__':
17:     print('단순 삽입 정렬을 수행합니다.')
18:     num = int(input('원소 수를 입력하세요.: '))
19:     x = [None] * num  # 원소 수가 num인 배열을 생성
20:
21:     for i in range(num):
22:         x[i] = int(input(f'x[{i}]: '))
23:
24:     insertion_sort(x)  # 배열 x를 단순 삽입 정렬
25:
26:     print('오름차순으로 정렬했습니다.')
27:     for i in range(num):
28:         print(f'x[{i}] = {x[i]}')
```

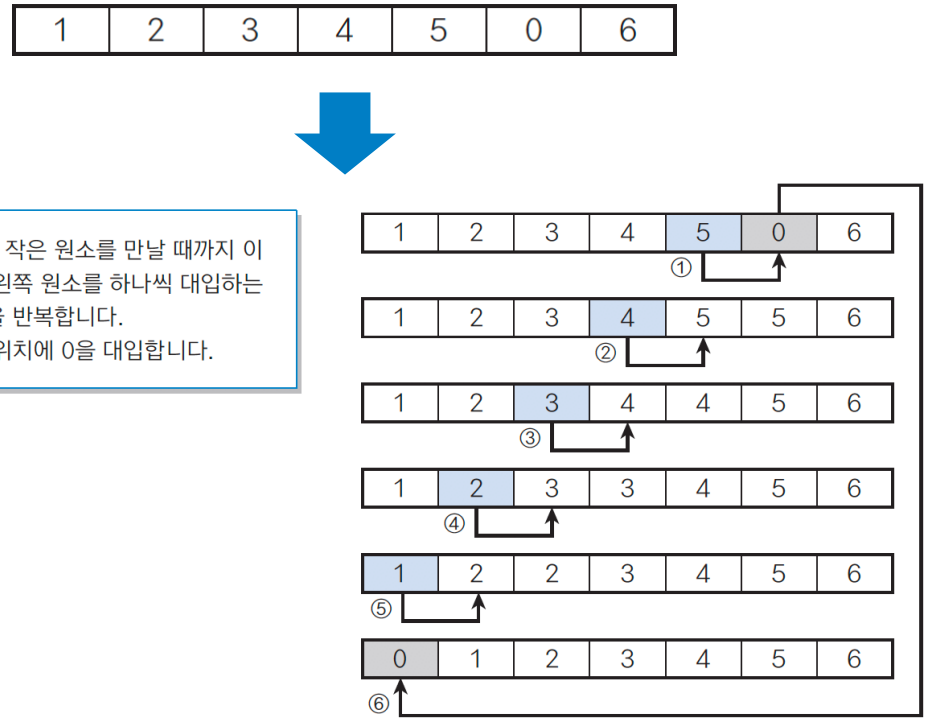
▶ 실행 결과

```
단순 삽입 정렬을 수행합니다.
원소 수를 입력하세요.: 7
x[0]: 6
x[1]: 4
x[2]: 3
x[3]: 7
x[4]: 1
x[5]: 9
x[6]: 8
오름차순으로 정렬했습니다.
x[0] = 1
x[1] = 3
x[2] = 4
x[3] = 6
x[4] = 7
x[5] = 8
x[6] = 9
```

06-5 셸 정렬

단순 삽입 정렬의 문제

- **단순 삽입 정렬의 한계**



- 여섯 번째 원소인 0을 삽입 정렬하려면 총 6번에 걸쳐 원소를 이동(대입)해야 함

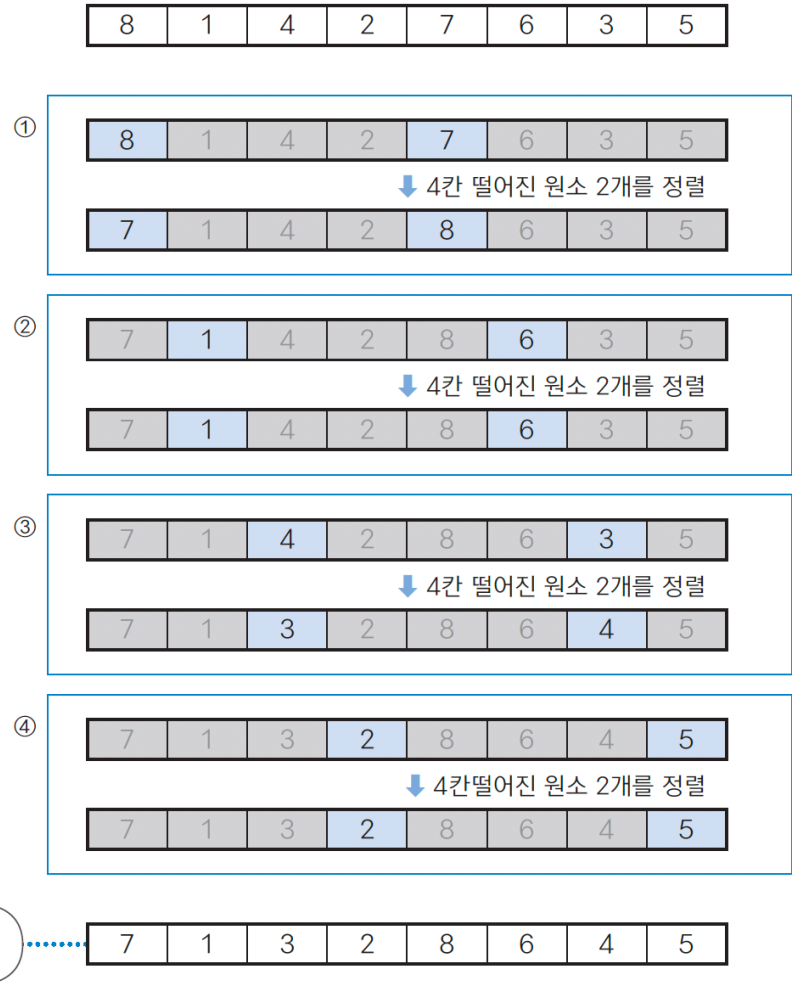
- 단순 삽입 정렬의 특징

  - **장점**: 이미 정렬을 마쳤거나 정렬이 거의 끝나가는 상태에서는 속도가 아주 빠름
  - **단점**: 삽입할 위치가 멀리 떨어져 있으면 이동 횟수가 많아짐

06-5 셸 정렬

셸 정렬 알아보기 – (1)

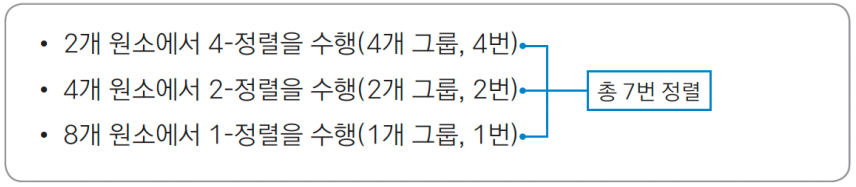
- **셸 정렬**shell sort
  - 단순 삽입 정렬의 장점을 살리면서 단점을 보완
  - 먼저 정렬할 배열의 원소를 그룹으로 나눠 각 그룹별로 정렬을 수행한 뒤 정렬된 그룹을 합치는 작업을 반복하여 원소의 이동 횟수를 줄이는 방법
  - 시간 복잡도는 $O(n^{1.25})$로 단순 정렬의 시간 복잡도인 $O(n^2)$보다 매우 빠르지만, 셸 정렬 알고리즘은 이웃하지 않고 떨어져 있는 원소를 서로 교환하므로 안정적인 정렬이 아님
  - 셸 정렬 과정에서 수행하는 각각의 정렬을 h-정렬 이라고 함
  - 4-정렬
    - 서로 4칸 떨어진 원소를 정렬하는 방법



06-5 셸 정렬

셸 정렬 알아보기 – (2)

- **셸 정렬**shell sort
  - 2-정렬
    - 2칸 떨어진 원소를 모두 꺼내 두 그룹으로 나누고 정렬을 수행
  - 마지막으로 배열 전체에 1-정렬 적용
  - 셸 정렬의 전체 흐름



06-5 셸 정렬

셸 정렬 알아보기 – (3)

- **실습 6-8**
  - 셸 정렬을 수행하는 프로그램

  - h의 초깃값은 n // 2
  - while 문을 반복할 때마다 다시 2로 나눈 값으로 업데이트
  - h의 변화

    - 원소 수가 8이면 4 → 2 → 1
    - 원소 수가 7이면 3 → 1

Do it! 실습 6-8 • 완성 파일 chap06/shell_sort1.py

```
01: # 셸 정렬 알고리즘 구현하기
02:
03: from typing import MutableSequence
04:
05: def shell_sort(a: MutableSequence) -> None:
06:     """셸 정렬"""
07:     n = len(a)
08:     h = n // 2
09:     while h > 0:
10:         for i in range(h, n):
11:             j = i - h
12:             tmp = a[i]
13:             while j >= 0 and a[j] > tmp:
14:                 a[j + h] = a[j]
15:                 j -= h
16:             a[j + h] = tmp
17:         h // = 2
18:
19: if __name__ == '__main__':
20:     print('셸 정렬을 수행합니다.')
21:     num = int(input('원소 수를 입력하세요.: '))
22:     x = [None] * num      # 원소 수가 num인 배열을 생성
23:
24:     for i in range(num):
25:         x[i] = int(input(f'x[{i}]: '))
26:
27:     shell_sort(x)         # 배열 x를 셸 정렬
28:
29:     print('오름차순으로 정렬했습니다.')
30:     for i in range(num):
31:         print(f'x[{i}] = {x[i]}')
```

▶ 실행 결과

```
셸 정렬을 수행합니다.
원소 수를 입력하세요.: 8
x[0]: 8
x[1]: 1
x[2]: 4
x[3]: 2
x[4]: 7
x[5]: 6
x[6]: 3
x[7]: 5
오름차순으로 정렬했습니다.
x[0] = 1
x[1] = 2
x[2] = 3
x[3] = 4
x[4] = 5
x[5] = 6
x[6] = 7
x[7] = 8
```

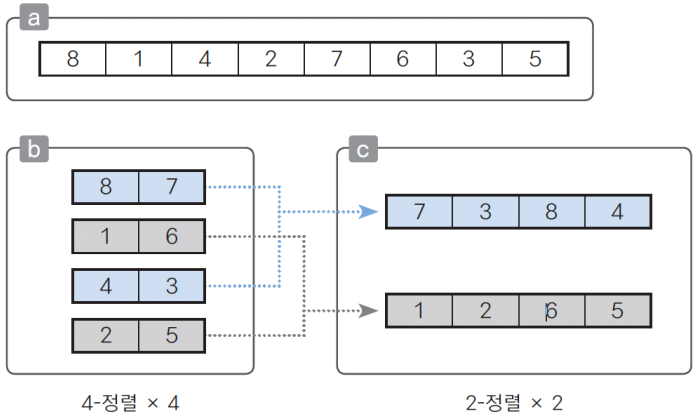
06-5 셸 정렬

셸 정렬 알아보기 – (4)

- **h값의 선택**
  - 원소 수인 n이 8이라면 h값을 다음과 같이 변화시킴

    h = 4 → 2 → 1

  - h값은 n부터 감소하다가 마지막에는 1이 됨
  - 배열 그룹을 나누는 과정



  - a 를 학생 8명의 점수라고 가정
  - b 처럼 학생 2명씩 4개 그룹으로 나누어 정렬
  - c 처럼 학생 4명씩 2개 그룹으로 나누어 다시 정렬
  - b 의 두 그룹을 합쳐서 c 가 되는 과정을 살펴보면, 파란색 그룹과 검은색 그룹은 섞이지 않음
  - 이렇게 두 그룹이 섞이지 않은 상태에서 c 로 합치면 다시 처음 단계인 a 와 같아짐
  - 애써 그룹으로 나누어서 정렬했지만, 충분히 그 기능을 하지 못한다는 것을 보여줌

- h값이 서로 배수가 되지 않아야 함

  h = … → 121 → 40 → 13 → 4 → 1

  - 1부터 시작하여 3배한 값에 1을 더하는 수열 사용

06-5 셸 정렬

셸 정렬 알아보기 – (5)

- **실습 6-9**
  - 1부터 시작하여 3배한 값에 1을 더하는 수열을 사용하여 셸 정렬을 수행하는 프로그램

  - 10~11행
    - h의 초깃값을 구함
    - 1부터 시작해서 h * 3 + 1의 수열을 사용하는 작업을 반복하지만 n // 9를 넘지 않는 최댓값을 h에 대입
    - h의 초깃값이 지나치게 크면 효과가 없기 때문에 배열의 원소 수인 n을 9로 나누었을 대 그 몫을 넘지 않도록 정해야 함
  - 13~21행
    - h값을 3으로 나누는 작업을 반복해서 결국에 h값은 1이 됨

Do it! 실습 6-9 • 완성 파일 chap06/shell_sort2.py

```
01: # 셸 정렬 알고리즘 구현하기(h * 3 + 1의 수열 사용)
02:
03: from typing import MutableSequence
04:
05: def shell_sort(a: MutableSequence) -> None:
06:     """셸 정렬(h * 3 + 1의 수열 사용)"""
07:     n = len(a)
08:     h = 1
09:
10:     while h < n // 9:
11:         h = h * 3 + 1
12:
13:     while h > 0:
14:         for i in range(h, n):
15:             j = i - h
16:             tmp = a[i]
17:             while j >= 0 and a[j] > tmp:
18:                 a[j + h] = a[j]
19:                 j -= h
20:             a[j + h] = tmp
21:         h //= 3
22:
23: if __name__ == '__main__':
24:     print('셸 정렬을 수행합니다(h * 3 + 1의 수열 사용).')
25:     num = int(input('원소 수를 입력하세요.: '))
26:     x = [None] * num      # 원소 수가 num인 배열을 생성
27:
28:     for i in range(num):
29:         x[i] = int(input(f'x[{i}]: '))
30:
31:     shell_sort(x)         # 배열 x를 셸 정렬
32:
33:     print('오름차순으로 정렬했습니다.')
34:     for i in range(num):
35:         print(f'x[{i}] = {x[i]}')
```

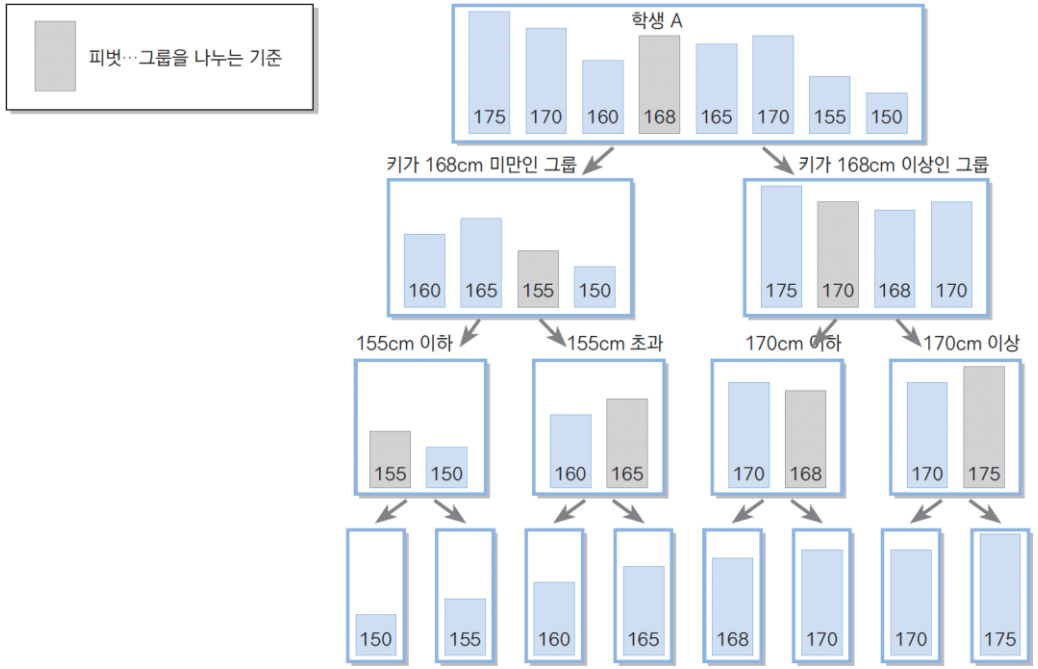
▶ 실행 결과

```
셸 정렬을 수행합니다(h * 3 + 1의 수열 사용).
원소 수를 입력하세요.: 8
x[0]: 8
x[1]: 1
x[2]: 4
x[3]: 2
x[4]: 7
x[5]: 6
x[6]: 3
x[7]: 5
오름차순으로 정렬했습니다.
x[0] = 1
x[1] = 2
x[2] = 3
x[3] = 4
x[4] = 5
x[5] = 6
x[6] = 7
x[7] = 8
```

06-6 퀵 정렬

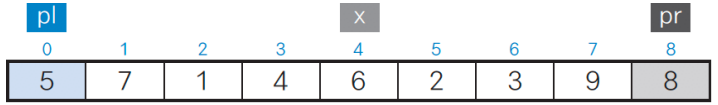
퀵 정렬 알아보기

- **퀵 정렬**quick sort
  - 일반적으로 사용되는 아주 빠른 정렬 알고리즘
  - 퀵 정렬의 예



배열을 두 그룹으로 나누기

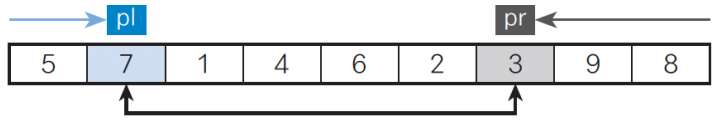
- **배열을 두 그룹으로 나누는 순서**



  - x: 피벗
  - pl: 왼쪽 끝 원소의 인덱스(왼쪽 커서)
  - pr: 오른쪽 끝 원소의 인덱스(오른쪽 커서)

- 피벗 이하인 원소를 배열 왼쪽으로, 피벗 이상인 원소를 배열 오른쪽으로 이동시키는 것이 목표

> - a[pl] >= x인 원소를 찾을 때까지 pl을 오른쪽 방향으로 스캔
> - a[pr] <= x인 원소를 찾을 때까지 pr를 왼쪽 방향으로 스캔

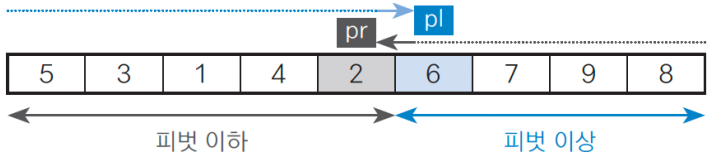


  - pl과 pr가 위치하는 원소 a[pl]과 a[pr]의 값을 교환

06-6 퀵 정렬

배열을 두 그룹으로 나누기 – (1)

- **배열을 두 그룹으로 나누는 순서**



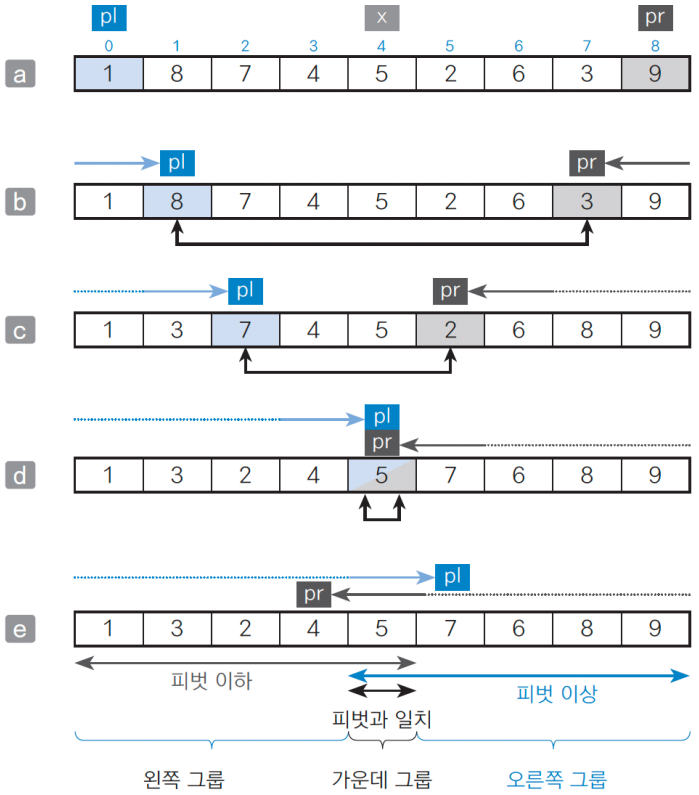
  - pl과 pr가 서로 교차하면 그룹 나누는 과정 끝

    - **피벗 이하인 그룹**: a[0], ..., a[pl - 1]
    - **피벗 이상인 그룹**: a[pr + 1], ..., a[n - 1]

  - 그룹을 나누는 작업이 끝난 다음 pl > pr + 1일 때에 한해서 다음과 같은 그룹이 만들어짐

    - **피벗과 일치하는 그룹**: a[pr + 1], ..., a[pl - 1]

- **피벗과 일치하는 그룹이 생성된 예**



06-6 퀵 정렬

배열을 두 그룹으로 나누기 – (2)

- **실습 6-10**: 배열을 두 그룹으로 나누는 프로그램 (배열 가운데에 있는 원소를 피벗으로 선택)

Do it! 실습 6-10

• 완성 파일 chap06/partition.py

```
01: # 배열을 두 그룹으로 나누기
02:
03: from typing import MutableSequence
04:
05: def partition(a: MutableSequence) -> None:
06:     """배열을 나누어 출력"""
07:     n = len(a)
08:     pl = 0          # 왼쪽 커서
09:     pr = n - 1      # 오른쪽 커서
10:     x = a[n // 2]  # 피벗(가운데 원소)
11:
12:     while pl <= pr:
13:         while a[pl] < x: pl += 1
14:         while a[pr] > x: pr -= 1
15:         if pl <= pr:
16:             a[pl], a[pr] = a[pr], a[pl]
17:             pl += 1
18:             pr -= 1
19:
20:     print(f'피벗은 {x}입니다.')
21:
22:     print('피벗 이하인 그룹입니다.')
23:     print(*a[0 : pl])               # a[0] ~ a[pl - 1]
25:     if pl > pr + 1:
26:         print('피벗과 일치하는 그룹입니다.')
27:         print(*a[pr + 1 : pl])      # a[pr + 1] ~ a[pl - 1]
28:
29:     print('피벗 이상인 그룹입니다.')
30:     print(*a[pr + 1 : n])           # a[pr + 1] ~ a[n - 1]
31:
32: if __name__ == '__main__':
33:     print('배열을 나눕니다.')
34:     num = int(input('원소 수를 입력하세요.: '))
35:     x = [None] * num                # 원소 수가 num인 배열을 생성
36:
37:     for i in range(num):
38:         x[i] = int(input(f'x[{i}]: '))
39:
40:     partition(x)                    # 배열 x를 나누어서 출력
```

배열 a를 피벗 x로 나누기

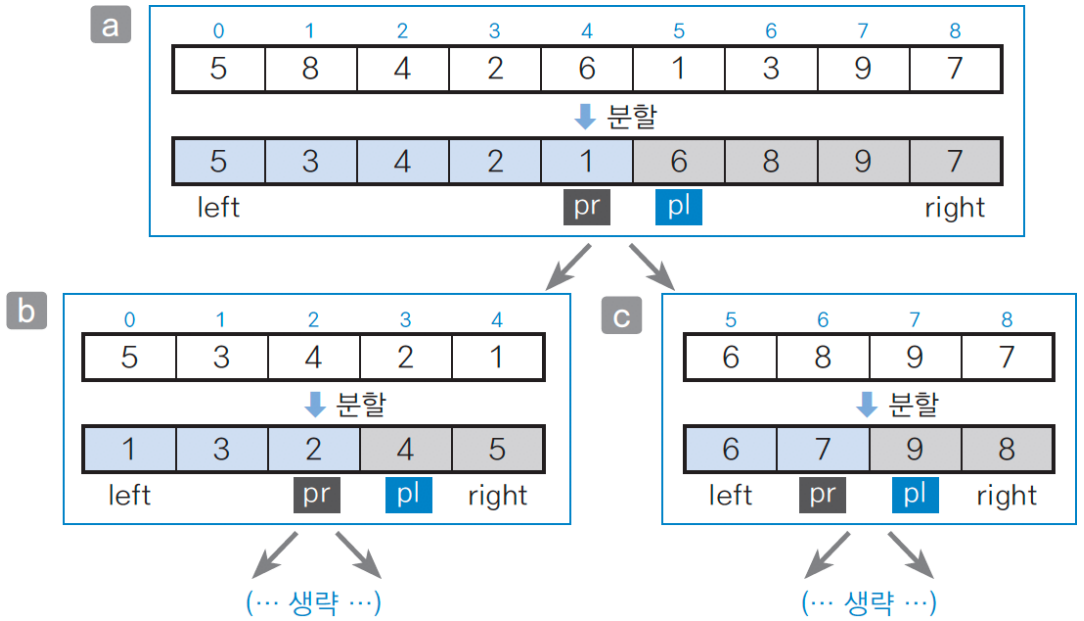
▶ 실행 결과

```
배열을 나눕니다.
원소 수를 입력하세요.: 9
x[0]: 1
x[1]: 8
x[2]: 7
x[3]: 4
x[4]: 5
x[5]: 2
x[6]: 6
x[7]: 3
x[8]: 9
피벗은 5입니다.
피벗 이하인 그룹입니다.
1 3 2 4 5
피벗과 일치하는 그룹입니다.
5
피벗 이상인 그룹입니다.
5 7 6 8 9
```

06-6 퀵 정렬

퀵 정렬 만들기 – (1)

- **퀵 정렬 알고리즘**
  - 퀵 정렬의 배열 나누기



- 원소 수가 2개 이상인 그룹을 반복해서 나누기

  - pr가 a[0]보다 오른쪽에 위치하면(left < pr) 왼쪽 그룹 나누기
  - pl이 a[8]보다 왼쪽에 위치하면(pl < right) 오른쪽 그룹 나누기

- 퀵 정렬은 8퀸 문제와 같은 분할 정복 알고리즘이므로 재귀 호출을 사용하여 구현 가능

- 퀵 정렬은 서로 이웃하지 않는 원소를 교환하므로 안정적이지 않은 알고리즘

06-6 퀵 정렬

퀵 정렬 만들기 – (2)

- **실습 6-11**
  - 퀵 정렬을 수행하는 프로그램
  - qsort( ) 함수: 배열 a와 배열을 나누는 구간의 첫 번째 원소(left), 마지막 원소(right)의 인덱스를 전달받아 퀵 정렬을 수행

Do it! 실습 6-11 • 완성 파일 chap06/quick_sort1.py

```
01: # 퀵 정렬 알고리즘 구현하기
02:
03: from typing import MutableSequence
04:
05: def qsort(a: MutableSequence, left: int, right: int) -> None:
06:     """a[left] ~ a[right]를 퀵 정렬"""
07:     pl = left                       # 왼쪽 커서
08:     pr = right                      # 오른쪽 커서
09:     x = a[(left + right) // 2]      # 피벗(가운데 원소)
10:
11:     while pl <= pr:
12:         while a[pl] < x: pl += 1
13:         while a[pr] > x: pr -= 1
14:         if pl <= pr:
15:             a[pl], a[pr] = a[pr], a[pl]
16:             pl += 1
17:             pr -= 1
18:
19:     if left < pr: qsort(a, left, pr)
20:     if pl < right: qsort(a, pl, right)
```

실습 6-10과 같음

```
22: def quick_sort(a: MutableSequence) -> None:
23:     """퀵 정렬"""
24:     qsort(a, 0, len(a) - 1)
25:
26: if __name__ == '__main__':
27:     print('퀵 정렬을 수행합니다.')
28:     num = int(input('원소 수를 입력하세요.: '))
29:     x = [None] * num      # 원소 수가 num인 배열을 생성
30:
31:     for i in range(num):
32:         x[i] = int(input(f'x[{i}]: '))
33:
34:     quick_sort(x)         # 배열 x를 퀵 정렬
35:
36:     print('오름차순으로 정렬했습니다.')
37:     for i in range(num):
38:         print(f'x[{i}] = {x[i]}')
```

▶ 실행 결과

```
퀵 정렬을 수행합니다.
원소 수를 입력하세요.: 9
x[0]: 5
x[1]: 8
x[2]: 4
x[3]: 2
x[4]: 6
x[5]: 1
x[6]: 3
x[7]: 9
x[8]: 7
오름차순으로 정렬했습니다.
x[0] = 1
x[1] = 2
x[2] = 3
x[3] = 4
x[4] = 5
x[5] = 6
x[6] = 7
x[7] = 8
x[8] = 9
```

06-6 퀵 정렬

비재귀적인 퀵 정렬 만들기 – (1)

- **실습 6-12**
  - qsort( ) 함수를 비재귀적으로 구현하는 프로그램
  - 데이터를 임시 저장하기 위해 스택 사용

> • **range**: 나눌 범위에서 맨 앞 원소의 인덱스와 맨 끝 원소의 인덱스를 조합한 튜플 스택

  - 스택의 크기는 right – left + 1이며 나누는 배열의 원소 수와 같음

Do it! 실습 6-12 • 완성 파일 chap06/quick_sort1_non_recur.py

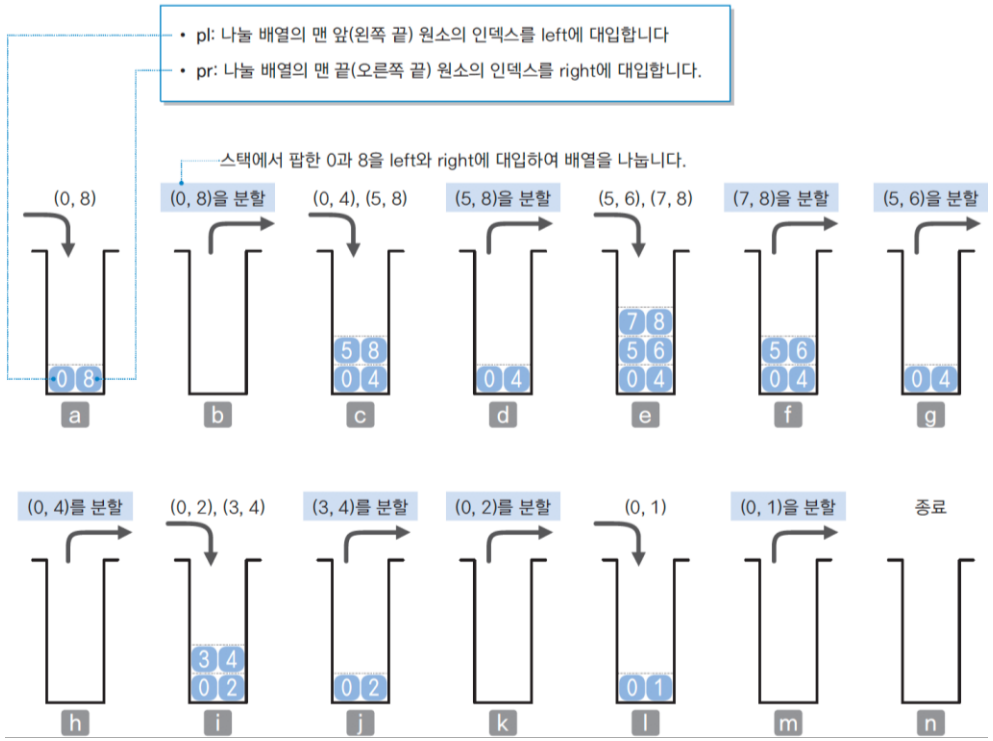
```
01: # 비재귀적인 퀵 정렬 구현하기
02:
03: from stack import Stack                          # 실습 4C-1 파일을 임포트
04: from typing import MutableSequence
05:
06: def qsort(a: MutableSequence, left: int, right: int) -> None:
07:     """a[left] ~ a [right]를 퀵 정렬(비재귀적인 퀵 정렬)"""
08:     range = Stack(right - left + 1)              # 스택 생성
09:
10:     range.push((left, right))
11:
12:     while not range.is_empty():
13:         pl, pr = left, right = range.pop()       # 왼쪽, 오른쪽 커서를 꺼냄
14:         x = a[(left + right) // 2]               # 피벗(가운데 원소)
15:
16:         while pl <= pr:
17:             while a[pl] < x: pl += 1
18:             while a[pr] > x: pr -= 1
19:             if pl <= pr:
20:                 a[pl], a[pr] = a[pr], a[pl]
21:                 pl += 1
22:                 pr -= 1
23:
24:         if left < pr: range.push((left, pr))     # 왼쪽 그룹의 커서를 저장
25:         if pl < right: range.push((pl, right))   # 오른쪽 그룹의 커서를 저장
(… 생략 …)
```

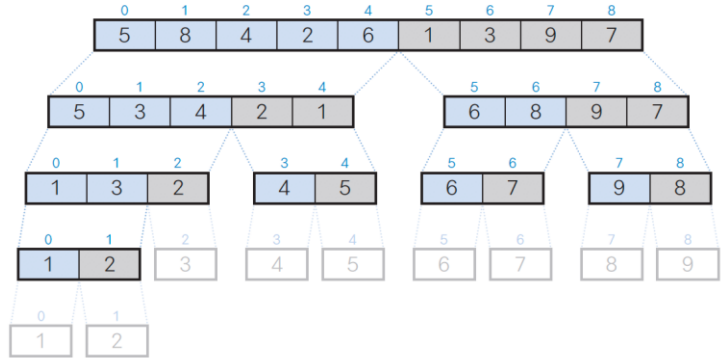
실습 6-10, 6-11과 같음

06-6 퀵 정렬

비재귀적인 퀵 정렬 만들기 – (2)

- **실습 6-12**
  - 스택의 변화



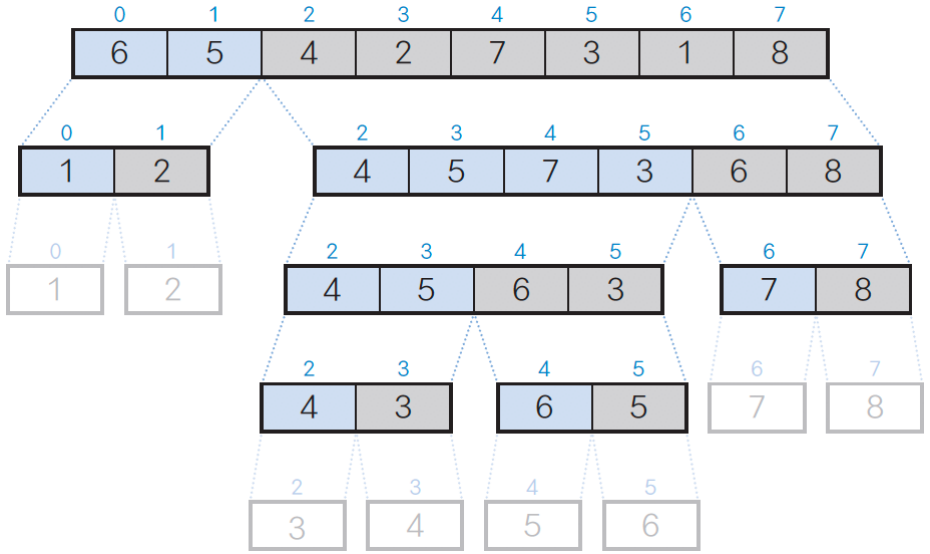


```
10:     range.push((left, right))
11:
12:     while not range.is_empty():
13:         pl, pr = left, right = range.pop()

(… 생략 …  a[left] ~ a[right]를 나누는 while 문)

24:         if left < pr: range.push((left, pr))
25:         if pl < right: range.push((pl, right))
```

06-6 퀵 정렬

비재귀적인 퀵 정렬 만들기 – (3)

- **스택의 크기**
  - 퀵 정렬의 배열 나누기 예
    - 피벗값은 2



- 배열을 스택에 푸시하는 순서를 정하는 규칙

> - **규칙 1**: 원소 수가 많은 쪽의 그룹을 먼저 푸시
> - **규칙 2**: 원소 수가 적은 쪽의 그룹을 먼저 푸시

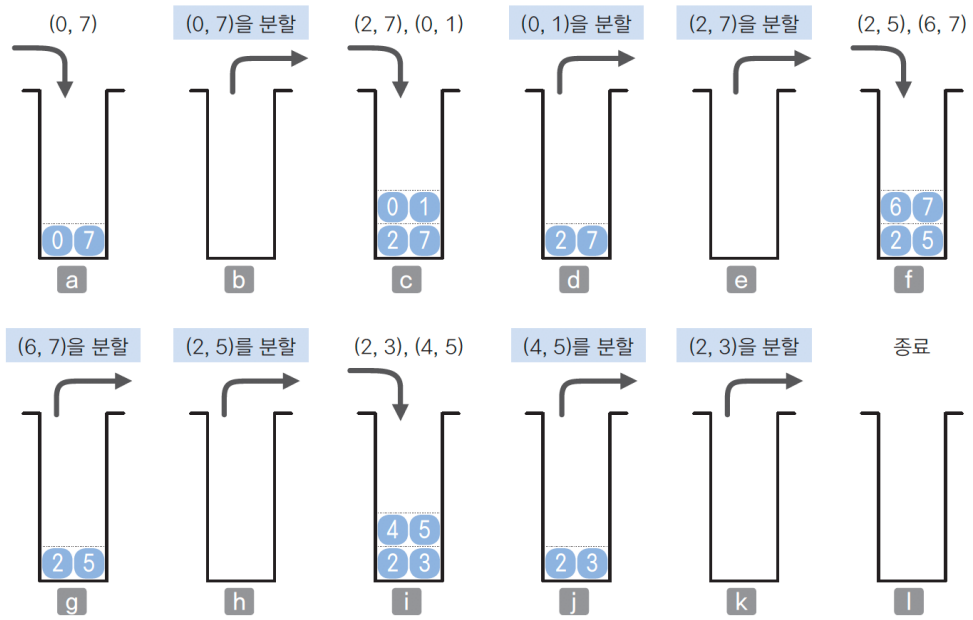
  - 일반적으로 원소 수가 적은 배열일수록 나누는 과정을 빠르게 마칠 수 있음
  - 규칙 1과 같이 원소 수가 많은 그룹의 나누기를 나중에 하고, 원소 수가 적은 그룹의 나누기를 먼저 하면 스택에 동시에 쌓이는 데이터 개수는 적어짐
  - 규칙 1, 2는 스택에 넣고 꺼내는 횟수(푸시, 팝)는 같지만, 동시에 쌓이는 데이터의 최대 개수는 다름

  - 규칙 1에서 배열의 원소 수가 n이면, 스택에 쌓이는 데이터의 최대 개수는 log n보다 적음.
    - 원소수 n이 100만 개라도 스택 최대 크기는 20이면 충분

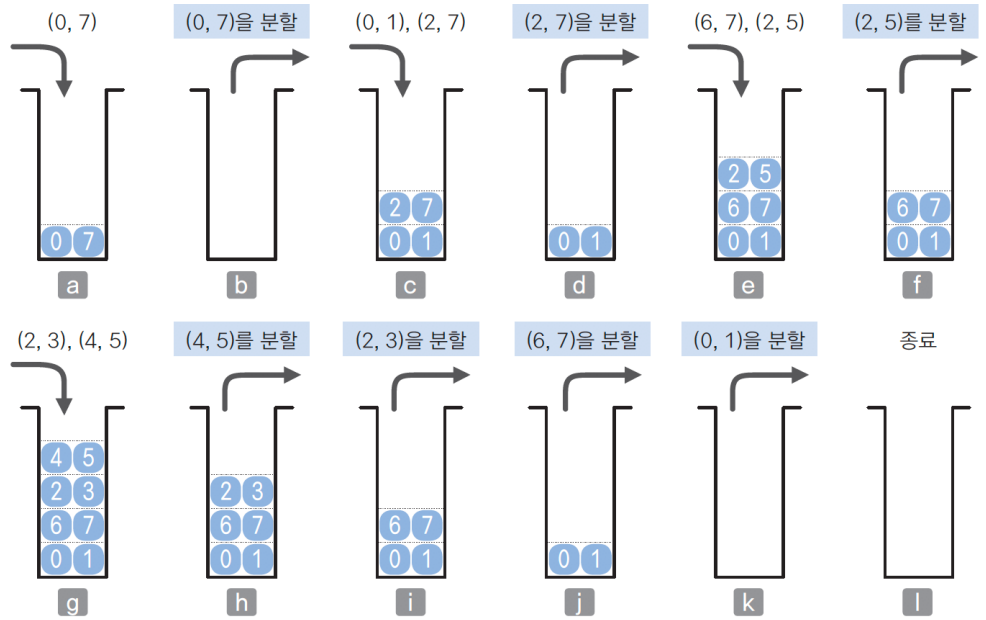
06-6 퀵 정렬

비재귀적인 퀵 정렬 만들기 – (4)

- **스택의 크기**
  - 규칙 1: 원소 수가 많은 그룹을 먼저 푸시
    - 예) a[0]~a[1]의 왼쪽 그룹과 a[2]~a[7]의 오른쪽 그룹 중 원소 수가 많은 그룹(2,7)을 먼저 푸시



  - 규칙 2: 원소 수가 적은 그룹을 먼저 푸시
    - 예) a[0]~a[1]의 왼쪽 그룹과 a[2]~a[7]의 오른쪽 그룹 중 원소 수가 많은 그룹(0, 1)을 먼저 푸시



06-6 퀵 정렬

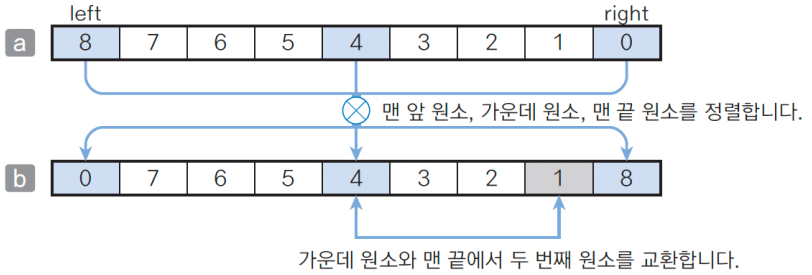
피벗 선택하기

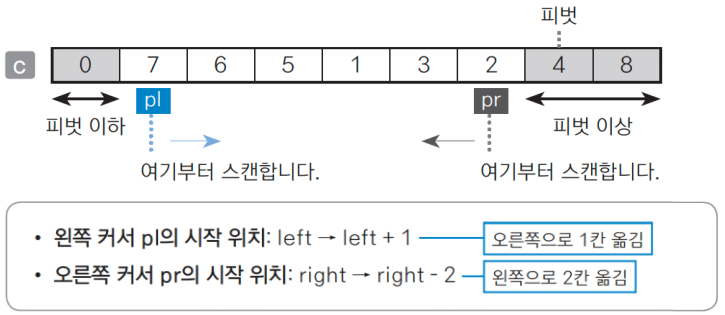
- **피벗 선택 방법**
  - 배열을 정렬한 뒤 가운데에 위치하는 값, 즉 전체에서 중앙값을 피벗으로 하는 것이 이상적!
  - 하지만 정렬된 배열의 중앙값을 구하려면 그에 대한 처리가 필요하고, 많은 계산 시간이 걸림

  - 피벗 선택 방법

- **방법 1**: 배열의 원소 수가 3 이상이면, 배열에서 임의의 원소 3개를 꺼내 중앙값인 원소를 피벗으로 선택
- **방법 2**: 배열의 맨 앞, 가운데, 맨 끝 원소를 정렬한 뒤 가운데 원소와 맨 끝에서 두 번째 원소를 교환. 맨 끝에서 두 번째 원솟값 a[right – 1]이 피벗으로 선택되고, 그 동시에 나눌 대상을 a[left + 1] ~ a[right– 2]로 좁힘

- 방법 2: 피벗 선택과 분할 범위 축소





- 이 방법을 사용하면 나누는 그룹이 한쪽으로 치우치는 것을 방지하고, 스캔할 원소를 3개 줄일 수 있음

06-6 퀵 정렬

퀵 정렬의 시간 복잡도

- **퀵 정렬의 시간 복잡도**
  - 배열을 조금씩 나누어 보다 작은 문제를 푸는 과정을 반복하므로 시간 복잡도는 O(n log n)
  - 정렬하는 배열의 초깃값이나 피벗을 선택하는 방법에 따라 실행 시간 복잡도가 증가하는 경우도 있음
    - 예) 매번 1개의 원소와 나머지 원소로 나누어진다면 n번의 분할이 필요. 최악의 경우 시간 복잡도는 $O(n^2)$
  - 퀵 정렬은 원소 수가 적은 경우는 그다지 빠른 알고리즘이 아님

- **실습 6-13**

  - 원소 수가 9개 미만인 경우 단순 삽입 정렬로 전환
  - 피벗 선택은 방법 2를 채택

  - **chap06/quick_sort2.py 참조** (p.272)