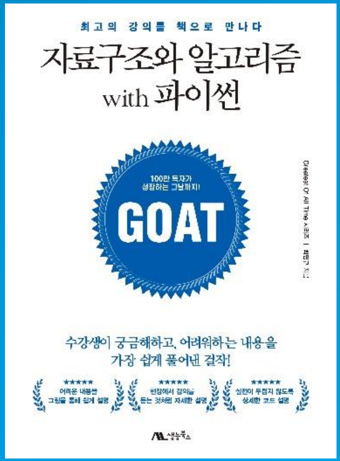
CHAPTER

06 정렬



SW알고리즘개발
9주차

6장. 정렬

06-1 정렬이란?

06-2 선택 정렬

06-3 삽입 정렬

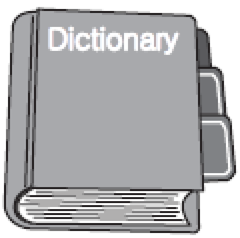
06-4 퀵 정렬

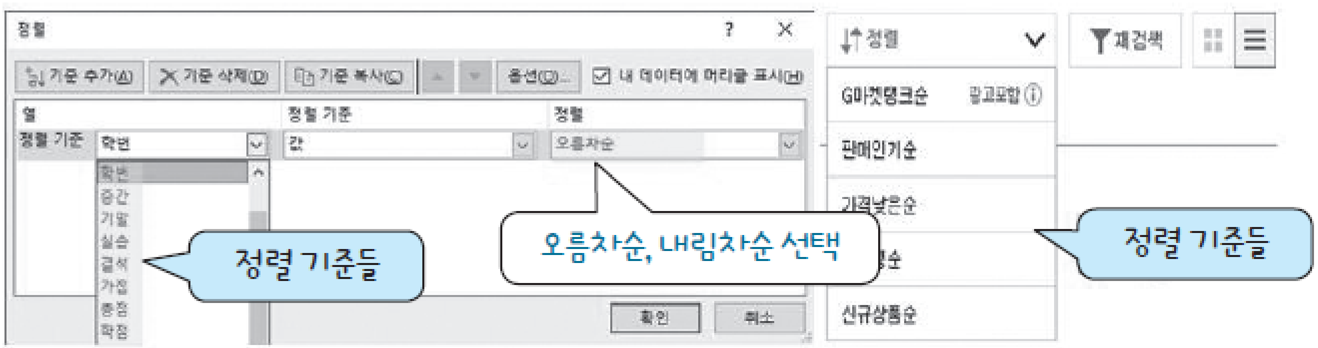
06-5 기수 정렬

06-6 파이썬의 정렬함수 활용하기

6.1 정렬이란?

- 정렬
  - 순서가 없는 사물들을 순서대로 나열하는 것

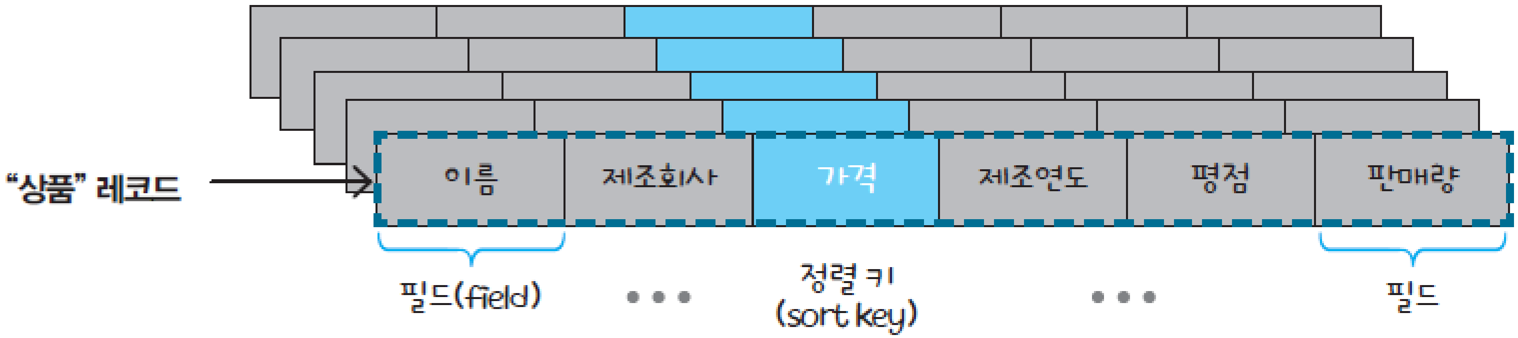




  - 정렬을 위해서는 사물들을 서로 비교할 수 있어야 함
  - 오름차순(ascending order)과 내림차순(descending order)

정렬 관련 용어

- 레코드(record), 필드(field), 키(key)



- 정렬이란 레코드들을 키(key)의 순서로 재배열하는 것

정렬의 분류

- 효율성
  - 비효율적: 삽입 정렬, 선택 정렬, 버블 정렬
  - 효율적: 퀵 정렬, 힙 정렬, 병합 정렬, 기수 정렬

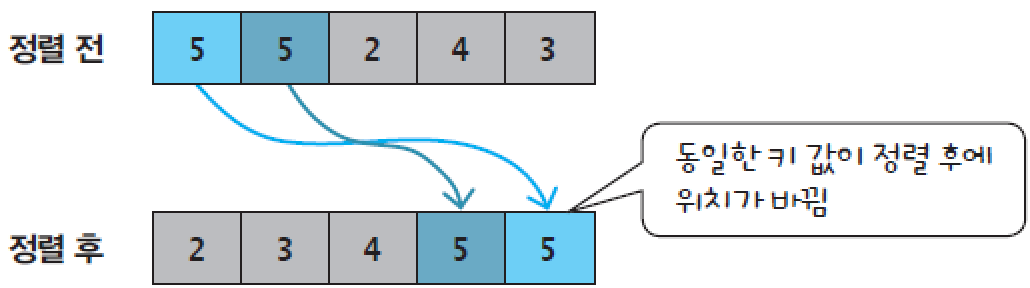
빨간 글씨는 수업에서 주로 사용하는 것

O(n²)

퀵 정렬은 O(n log n), O(n²)이다.

O(nk)

- 안정성



- 제자리 정렬
  - 입력 배열 이외에 추가적인 배열을 사용하지 않는 정렬

6.2 선택 정렬(Selection Sort)

- 가장 단순하지만 비효율적 방법
- 교환 연산 = 비교 연산
- 제자리 정렬 방식

입력 리스트

선택 정렬 알고리즘(제자리 정렬 방식)

```
def selection_sort(A) :
    n = len(A)                                   # 리스트의 크기
    for i in range(n-1) :
        least = i
        for j in range(i+1, n) :
            if (A[j]<A[least]) :
                least = j
        A[i], A[least] = A[least], A[i]          # A[i]와 A[least] 교환
```

i는 정렬되지 않은 부분의 시작 인덱스
0부터 n-2까지 순서대로 대입

i+1부터 n-1까지의 요소 중에서 최솟값의
인덱스 least를 구함

```
Original  : [6, 3, 7, 4, 9, 1, 5, 2, 8]
  Step  1 = [1, 3, 7, 4, 9, 6, 5, 2, 8]
  Step  2 = [1, 2, 7, 4, 9, 6, 5, 3, 8]
  Step  3 = [1, 2, 3, 4, 9, 6, 5, 7, 8]
  Step  4 = [1, 2, 3, 4, 9, 6, 5, 7, 8]
  Step  5 = [1, 2, 3, 4, 5, 6, 9, 7, 8]
  Step  6 = [1, 2, 3, 4, 5, 6, 9, 7, 8]
  Step  7 = [1, 2, 3, 4, 5, 6, 7, 9, 8]
  Step  8 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
Selection : [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

정렬 안된 부분

정렬된 부분

선택 정렬의 특징(1)

- 성능
  - 비교연산 : 주어진 입력 리스트에서 매 단계마다 최소 값을 찾아야 함
    - n 개의 요소가 있을 때, 외부 루프는 $n-1$ 번 반복, 내부 루프는 각 단계에서 $n-1, n-2, n-3, \dots, 1$ 번 반복
    - 연산의 총 횟수 = $(n-1)+(n-1)+ \dots +1 = \frac{n(n-1)}{2} \in O(n^2)$
  - 교환 연산 : 각 단계마다 최소 값을 찾은 후, 그 위치에 있는 요소와 교환
    - 리스트가 이미 정렬되어 있어도 각 요소를 한 번씩 교환하게 되므로, 최악의 경우와 최선의 경우 모두 $n-1$ 번 교환이 발생
    - 연산의 총 횟수 = $O(n)$
  - 전체 시간 복잡도 : $O(n^2)$ - 최선, 평균, 최악의 경우에 모두 동일
- 특징
  - 요소의 개수가 적거나, 데이터가 거의 정렬되어 있지 않은 경우 사용
  - 큰 데이터에 대해서는 비효율적
  - 제자리 정렬
  - 자료의 구성에 상관없이 연산의 횟수가 결정됨

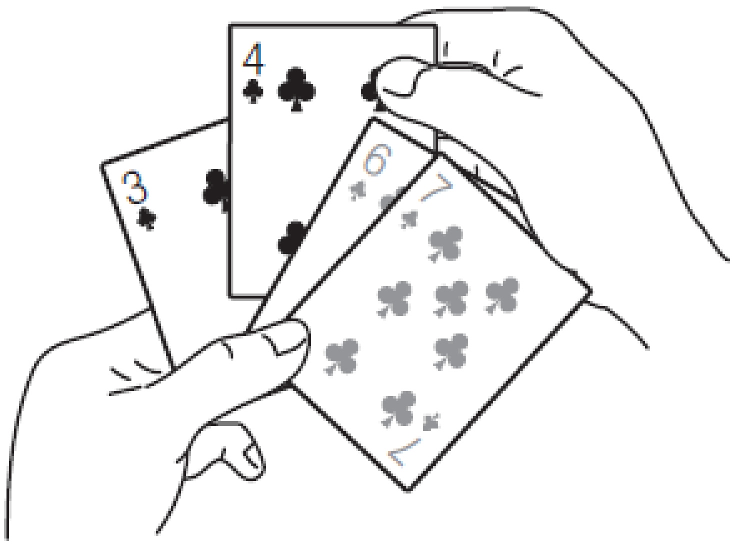
선택 정렬의 특징(2)

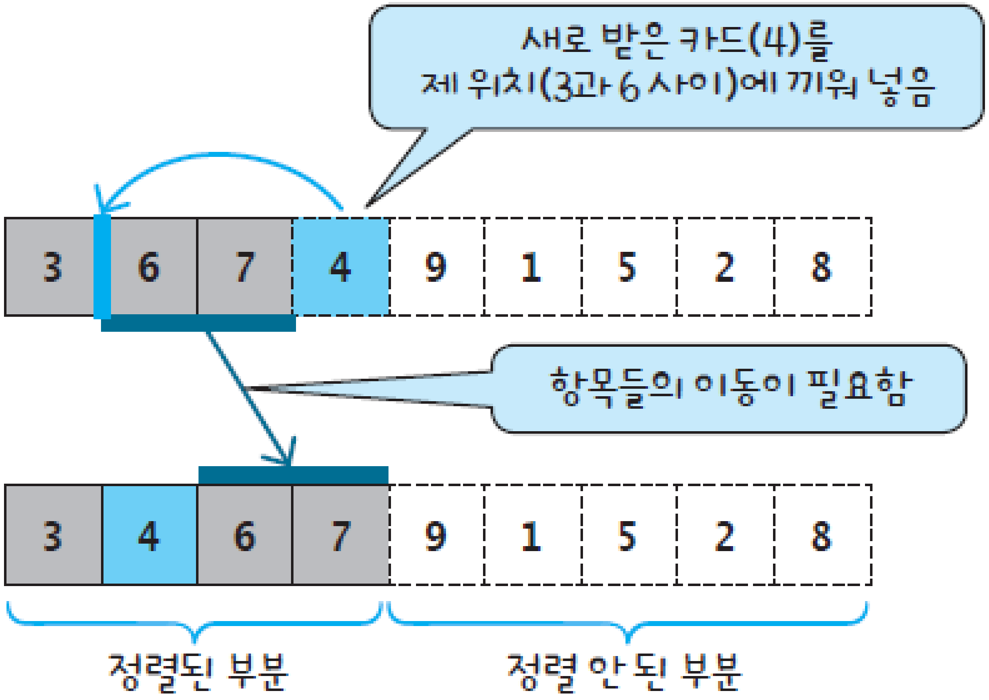
- 특징
  - 안정적인 정렬 알고리즘이 아님
  - 중복된 값이 있을 때 원래의 순서를 유지하지만, 이는 동일한 값을 가진 원소들의 상대적인 순서가 정렬 후에 바뀔 수 있음을 의미
  - 예: 선택 정렬 과정에서 특정한 구현에서는 두 값을 교환할 수 있는 상황이 발생할 수 있으며, 이는 안정성을 보장하지 못함

```
Before sorting: [(3, 'a'), (1, 'b'), (4, 'c'), (1, 'd'), (5, 'e'), (9, 'f'), (2, 'g'), (6, 'h'), (5, 'i'), (3, 'j')]
  Step  1 = [(1, 'b'), (3, 'a'), (4, 'c'), (1, 'd'), (5, 'e'), (9, 'f'), (2, 'g'), (6, 'h'), (5, 'i'), (3, 'j')]
  Step  2 = [(1, 'b'), (1, 'd'), (4, 'c'), (3, 'a'), (5, 'e'), (9, 'f'), (2, 'g'), (6, 'h'), (5, 'i'), (3, 'j')]
  Step  3 = [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (5, 'e'), (9, 'f'), (4, 'c'), (6, 'h'), (5, 'i'), (3, 'j')]
  Step  4 = [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (5, 'e'), (9, 'f'), (4, 'c'), (6, 'h'), (5, 'i'), (3, 'j')]
  Step  5 = [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (3, 'j'), (9, 'f'), (4, 'c'), (6, 'h'), (5, 'i'), (5, 'e')]
  Step  6 = [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (3, 'j'), (4, 'c'), (9, 'f'), (6, 'h'), (5, 'i'), (5, 'e')]
  Step  7 = [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (3, 'j'), (4, 'c'), (5, 'i'), (6, 'h'), (9, 'f'), (5, 'e')]
  Step  8 = [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (3, 'j'), (4, 'c'), (5, 'i'), (5, 'e'), (9, 'f'), (6, 'h')]
  Step  9 = [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (3, 'j'), (4, 'c'), (5, 'i'), (5, 'e'), (6, 'h'), (9, 'f')]
After sorting: [(1, 'b'), (1, 'd'), (2, 'g'), (3, 'a'), (3, 'j'), (4, 'c'), (5, 'i'), (5, 'e'), (6, 'h'), (9, 'f')]
```

6.3 삽입 정렬(Insertion Sort)

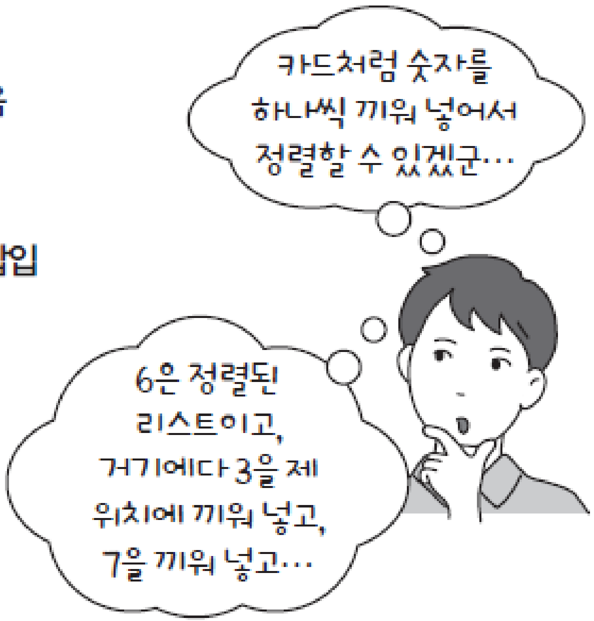
- 카드를 정렬하는 방법과 유사





삽입 정렬 과정

6	3	7	4	9	1	5	2	8	초기 상태. 6은 정렬된 리스트
6	3	7	4	9	1	5	2	8	3을 정렬된 리스트에 삽입
3	6	7	4	9	1	5	2	8	7은 이미 제자리에 있음
3	6	7	4	9	1	5	2	8	4를 정렬된 리스트에 삽입
3	4	6	7	9	1	5	2	8	9는 제자리에 있음
				⋮					⋮
1	2	3	4	5	6	7	8	9	정렬 완료



정렬된 부분 리스트에 항목을 끼워 넣는 과정

삽입 정렬 알고리즘

```python
def insertion_sort(A) :
    n = len(A)
    for i in range(1, n) :          # i범위: 1~n-1
        key = A[i]
        j = i-1
        while j>=0 and A[j] > key :
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key              # j+1이 A[i]가 삽입될 위치임
```

삽입할 요소를 미리 key에 저장해 둠

i-1요소부터 비교하여 앞으로 진행하는데,
이 요소가 key보다 크면 뒤로 한 칸 옮김

```
Original  : [6, 3, 7, 4, 9, 1, 5, 2, 8]
  Step  1 = [3, 6, 7, 4, 9, 1, 5, 2, 8]
  Step  2 = [3, 6, 7, 4, 9, 1, 5, 2, 8]
  Step  3 = [3, 4, 6, 7, 9, 1, 5, 2, 8]
  Step  4 = [3, 4, 6, 7, 9, 1, 5, 2, 8]
  Step  5 = [1, 3, 4, 6, 7, 9, 5, 2, 8]
  Step  6 = [1, 3, 4, 5, 6, 7, 9, 2, 8]
  Step  7 = [1, 2, 3, 4, 5, 6, 7, 9, 8]
  Step  8 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
Selection : [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

정렬 안된 부분

정렬된 부분

끼워 넣은 항목들

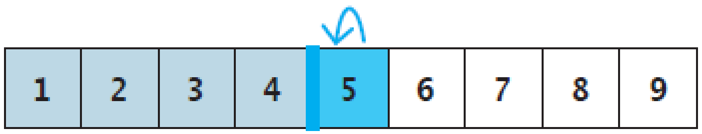
삽입 정렬의 시간복잡도의 효율성

- 삽입 정렬의 시간 복잡도는 데이터가 정렬된 정도에 따라 달라짐
- **최선의 경우 : $\boldsymbol{O(n)}$**
  - **이미 오름차순으로 정렬된 배열을 정렬:** 각 요소는 한 번의 비교만 필요
  - **비교 연산 횟수**: $n-1$회로, $O(n)$
  - **교환 연산 횟수**: 이동이 필요 없으므로, 교환 연산은 발생하지 않음

    ㄴ 이동 연산 발생 x
- **평균의 경우** : $O(n^2)$
  - **데이터가 무작위로 분포:** 각 요소를 정렬된 위치에 삽입해야 하므로, 평균적으로 전체 길이의 절반만큼 비교가 필요
  - **비교와 이동 연산**: 평균적으로 $n/2$번의 비교와 교환이 필요
  - $n \times (n/2) = O(n^2)$
- **최악의 경우** : $O(n^2)$
  - **데이터가 이미 내림차순으로 정렬된 배열을 정렬**: 각 요소를 처음 위치까지 이동시켜야 하므로 비교와 교환 횟수가 최대가 됨.
  - **비교와 이동 연산 횟수**: 첫 번째 요소에서 마지막 요소까지 $n-1, n-2, \dots, 1$회 비교가 필요
    - $(n-1)+(n-1)+\ \dots\ +1 = \frac{n(n-1)}{2} \in O(n^2)$

삽입 정렬의 특징

- 최선, 최악, 평균의 경우 효율이 다름
  - 최선: 오름차순으로 정렬된 리스트 →$O(n)$



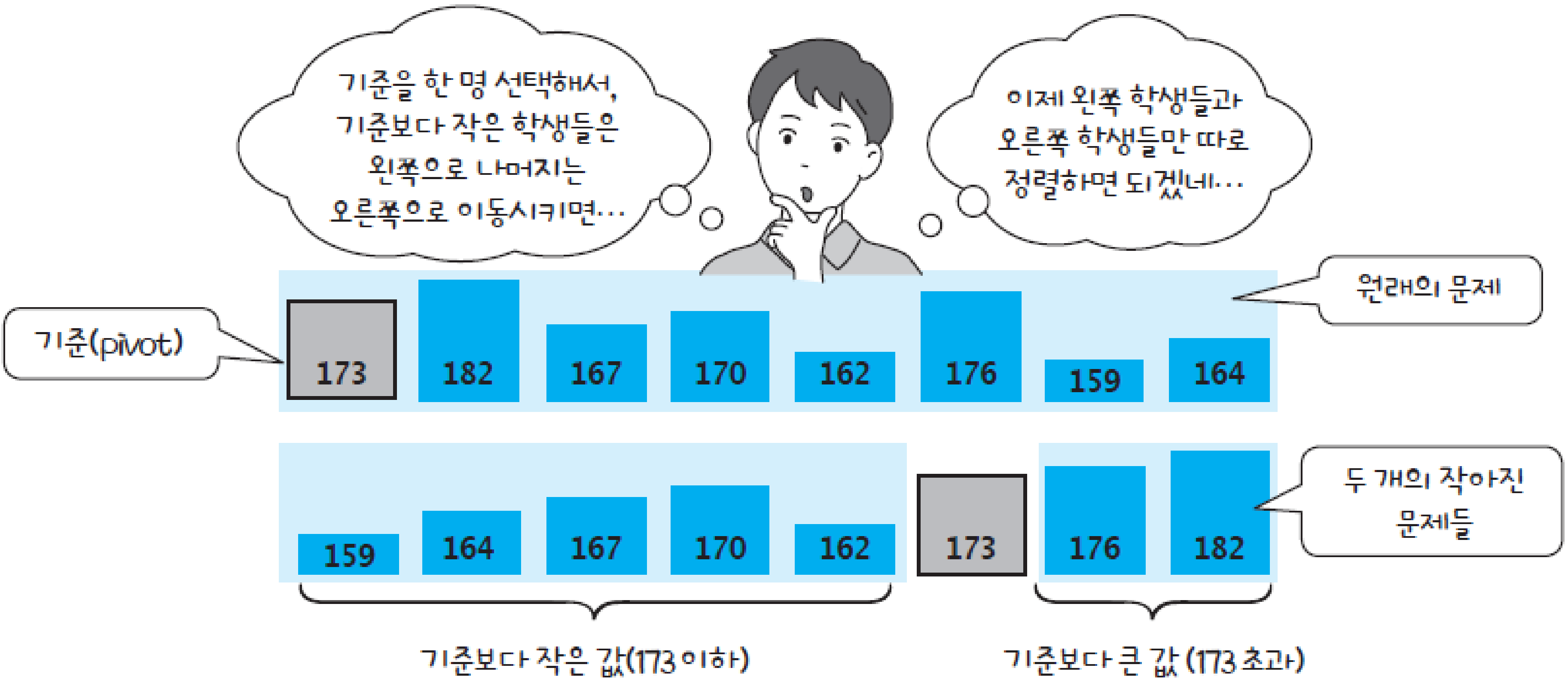
한 번만 비교하면 끼워 넣기 완료

  - 최악: 역순으로 정렬된 리스트 → $O(n^2)$

모든 항목을 검사하고 맨 앞에 끼워 넣어야 함

- 특징
  - 효율적이지 않음.
  - 많은 레코드의 이동.
  - 제자리 정렬. 안정성 충족
  - 레코드 대부분이 이미 정렬된 경우라면 효율적으로 사용됨

6.4 퀵 정렬(Quick Sort)



퀵 정렬 과정

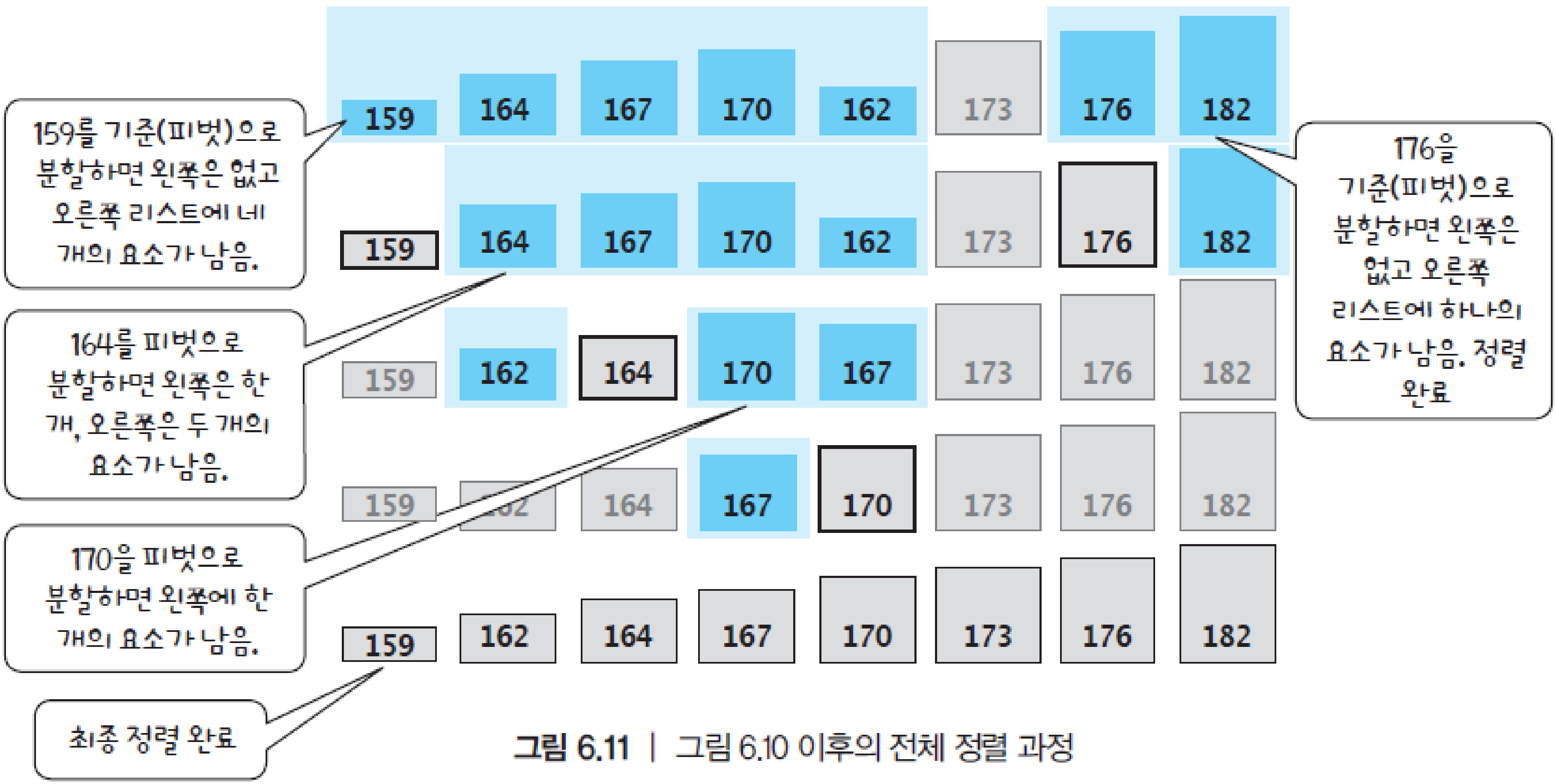


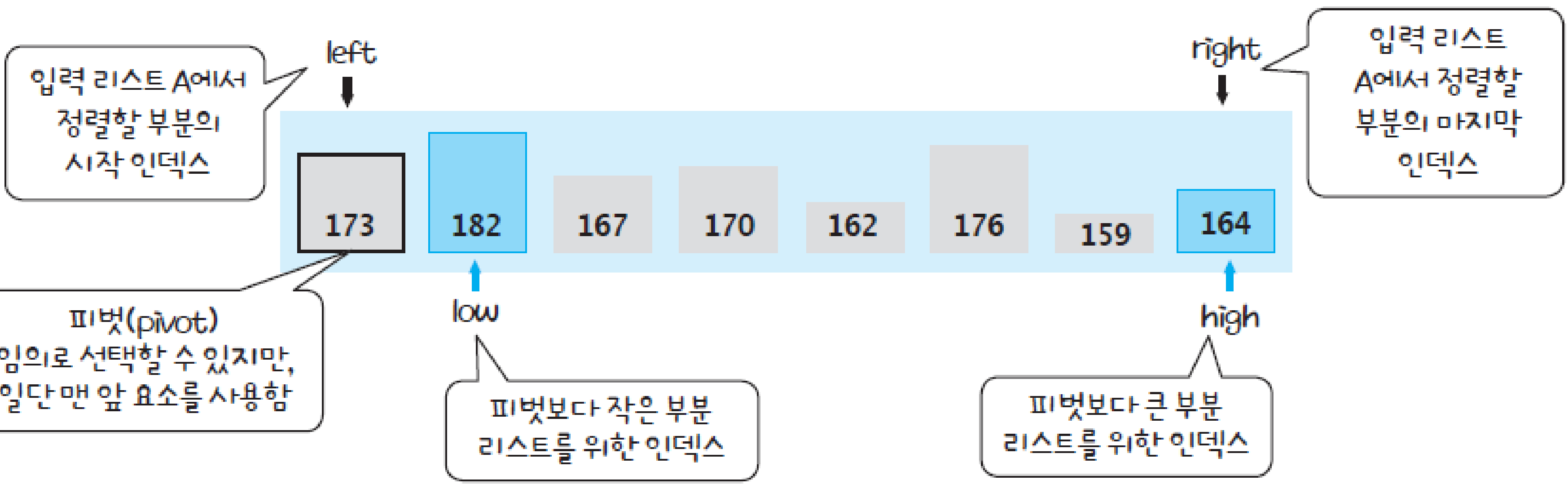
그림 6.11 | 그림 6.10 이후의 전체 정렬 과정

퀵 정렬 알고리즘

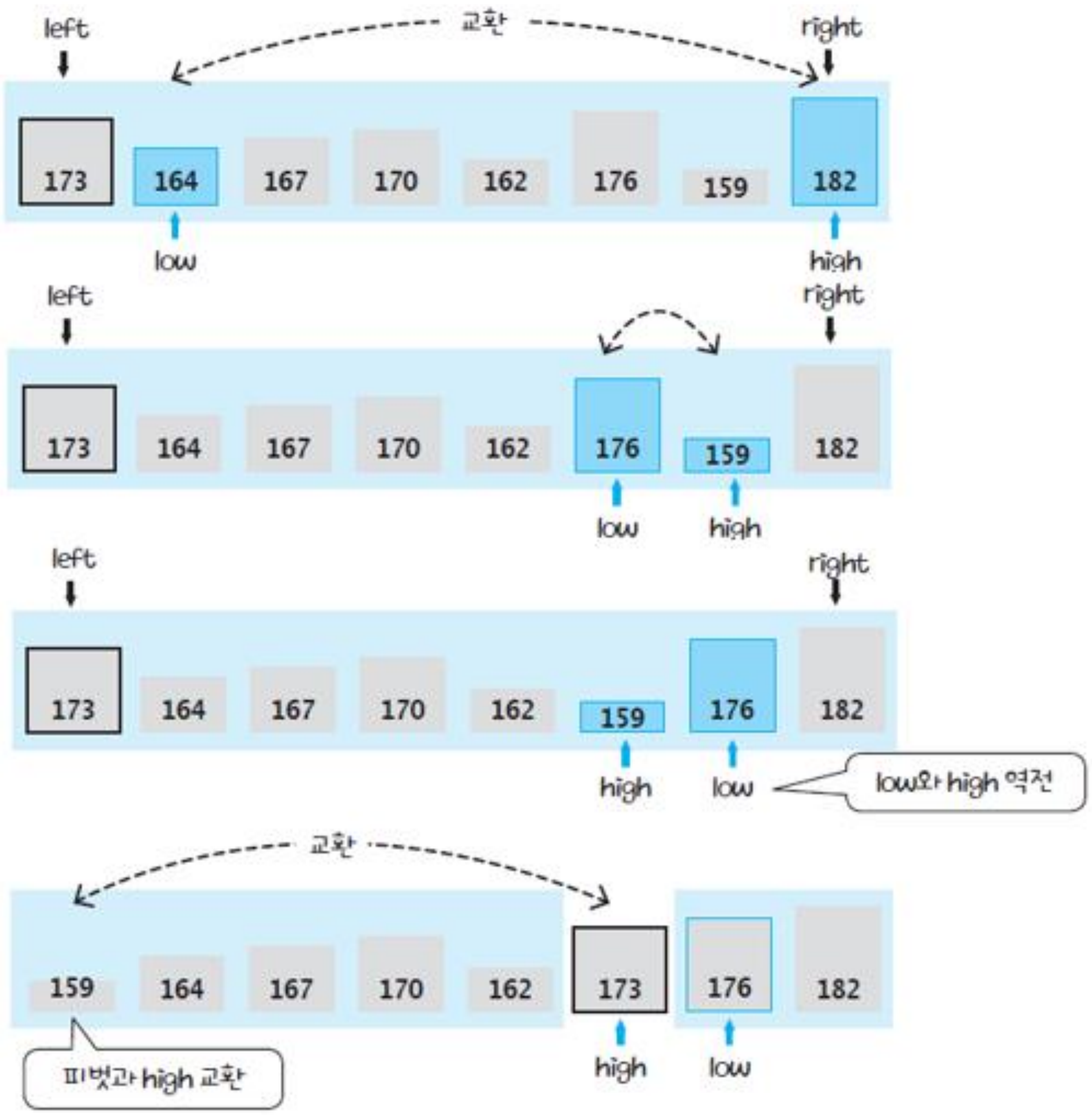
```
def quick_sort(A, left, right) :
    if left<right :                # 정렬 범위가 2개 이상인 경우
        q = partition(A, left, right)
        quick_sort(A, left, q - 1)
        quick_sort(A, q + 1, right)
```

피벗을 중심으로 리스트를 두 부분으로 분할하고, 피벗의 위치 q를 구함.
이제 왼쪽(left~q-1)과 오른쪽(q+1~right) 부분 리스트를 각각 정렬하면 전체 정렬 완료.

- 분할 알고리즘



분할 과정


left
교환
right
173
164
167
170
162
176
159
182
low
high
left
high
right
173
164
167
170
162
176
159
182
low
high
left
right
173
164
167
170
162
159
176
182
high
low
low와 high 역전
교환
159
164
167
170
162
173
176
182
피벗과 high 교환
high
low

분할 알고리즘

```
def partition(A, left, right) :
    pivot = A[left]
    low = left + 1
    high = right

    while (low < high) :   # low와 high가 역전되지 않는 한 반복
        while low <= right and A[low] <= pivot :
            low += 1       # A[low]<=피벗이면 low를 오른쪽으로 진행

        while high >= left and A[high] > pivot :
            high-= 1       # A[high]>피벗이면 high를 왼쪽으로 진행

        if low < high :    # 역전이 아니면 두 레코드 교환
            A[low], A[high] = A[high], A[low]

    A[left], A[high] = A[high], A[left]
    return high
```

왼쪽(left) 요소를 피벗으로 사용하면, low는 left+1이 되고, high는 right가 됨.

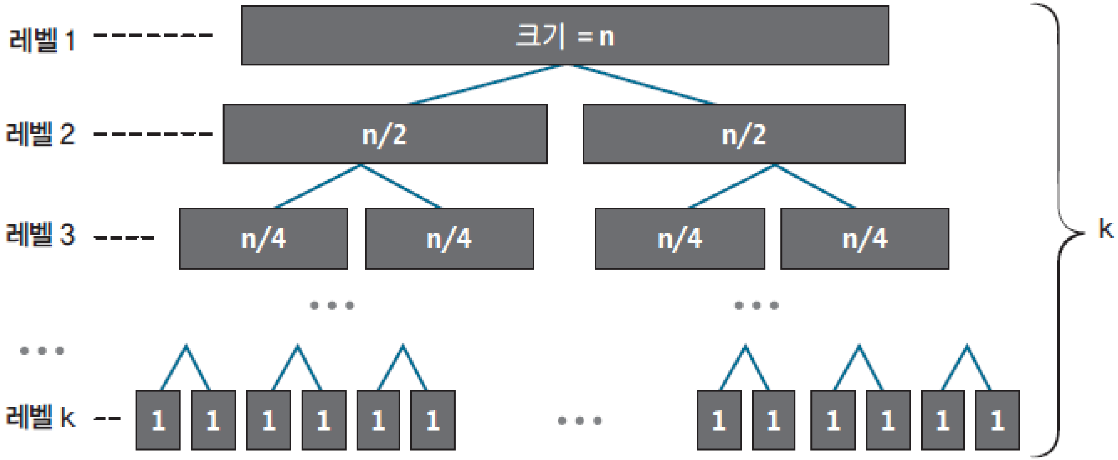
양쪽에서 조건에 맞지 않는 요소를 찾는 과정

마지막으로 피벗과 high를 교환하고, 피벗의 인덱스 high를 반환

퀵 정렬의 성능

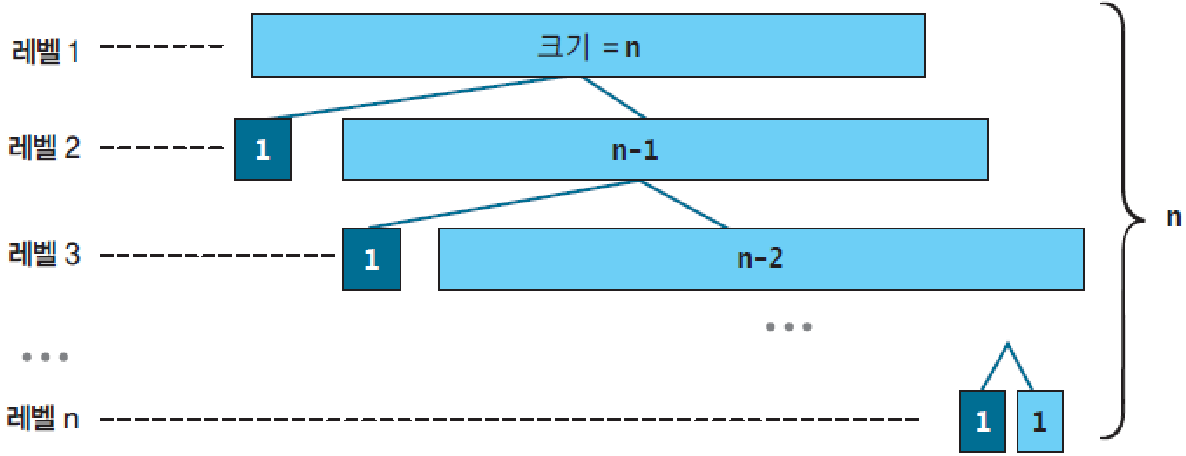
- 최선의 분할

$O(n\ log\ n)$



- 최악의 분할

$O(n^2)$



퀵 정렬의 시간복잡도

- 피벗 선택 방식과 데이터의 분포에 따라 크게 달라짐
- 최선의 경우 : $O(nlogn)$
  - **조건**: 매번 피벗이 리스트를 정확히 반으로 나누는 경우
  - **분할 과정**: 각 분할 단계에서 리스트를 절반씩 나누어 처리하므로 분할의 깊이는 $\log n$
  - **비교 연산 횟수**: 각 단계에서 최대 n번의 비교가 필요하므로 전체 비교 횟수는 $n \times \log n$
- 평균의 경우: $O(nlogn)$
  - **조건**: 피벗이 리스트의 중앙에 가깝게 선택되는 경우
  - **분할 과정**: 리스트의 크기는 균등하지 않더라도, 피벗이 중앙 근처에 있을 때는 여전히 각 단계에서 n번의 비교가 필요하고, 분할 깊이는 $logn$에 가깝다.
- 최악의 경우: $O(n^2)$
  - **조건**: 피벗이 항상 리스트의 최댓값 또는 최솟값으로 선택되는 경우
    - 이미 정렬된 리스트에서 첫 번째 요소를 피벗으로 선택하면 발생
  - **분할 과정**: 매 분할 단계에서 한쪽 서브리스트만 생성되고, 크기는 매번 1씩 감소함. 분할 깊이는 최대 n에 이름.
  - **비교 연산 횟수**: 각 단계에서 최대 n번의 비교가 필요하므로, 총 비교 연산 횟수는 $n * n = n^2$

퀵 정렬의 추가 최적화 방법

- 퀵 정렬은 피벗을 잘 선택하면 최악의 경우를 피할 수 있어 실제로는 $O(\mathrm{n}\log n)$에 가까운 성능을 가짐
- 20세기 과학기술에 가장 큰 영향을 준 10대 알고리즘
  - 불필요한 데이터의 이동을 줄임
  - 먼 거리의 데이터를 교환
  - 한번 결정된 피벗들이 추후 연산에서 제외
  - 평균적으로 매우 빠른 정렬 알고리즘
  - 특정 경우(특히 이미 정렬된 배열을 처리할 때)에는 성능이 크게 저하
  - 최악의 경우를 피하는 방법 : 평균 $O(n\log n)$에 유지
    - **랜덤 피벗 선택**: 피벗을 무작위로 선택하여 최악의 경우 발생 확률을 감소
    - **median-of-three**: 리스트의 시작, 중간, 끝 요소 중 중간값을 피벗으로 선택해 비교적 중앙에 가까운 피벗을 선택

예: 퀵 정렬에서 최악의 경우

- 퀵 정렬의 핵심은 피벗을 선택한 후, 배열을 피벗을 기준으로 분할
- 입력 배열 구성: 피벗을 배열의 처음이나 끝에서 선택할 경우, 특히 이미 정렬된 배열이나 역순 배열을 처리할 때 분할이 매우 비효율적으로 이루어져 시간 복잡도가 $O(n^2)$.
- 해결책: Median of Three
  - 세 가지 값을 선택한 후, 이 중 중간값을 피벗으로 사용하는 것
  - 중간값을 선택함으로써 피벗이 배열의 중앙값에 가깝게 위치할 가능성이 높아지고, 분할이 더 균형적으로 이루어지도록 함
  - 알고리즘:
    - 배열의 첫 번째 값, 중간 값, 마지막 값을 선택
    - 이 세 값의 중앙값(즉, 세 값 중 크기 순서로 두 번째로 큰 값)을 피벗으로 사용

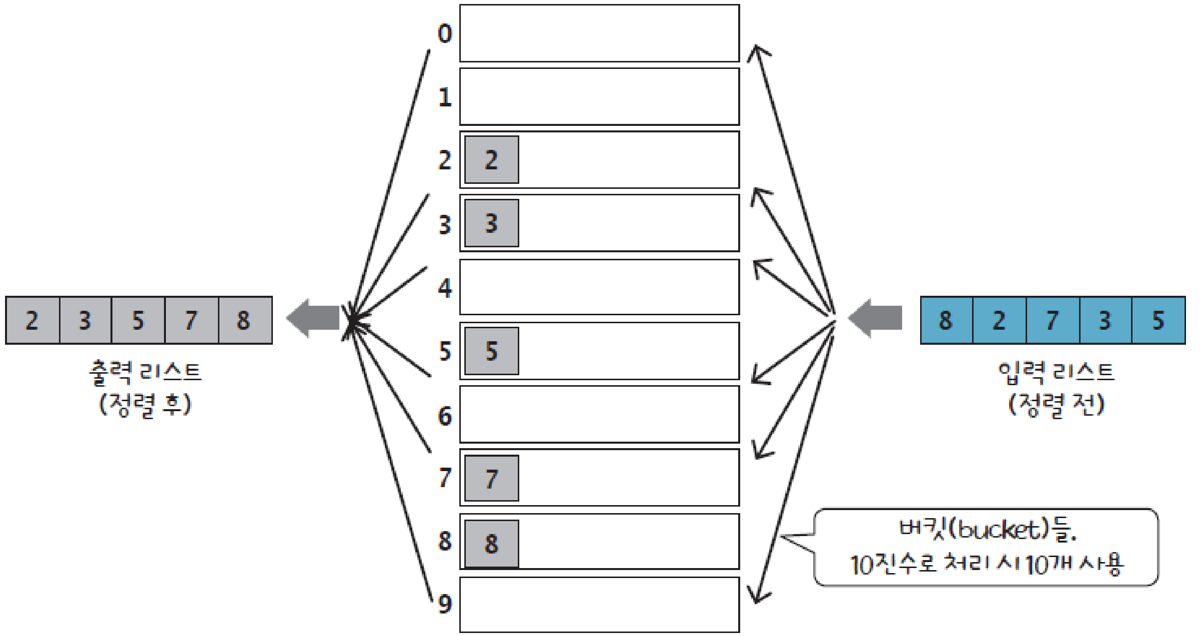
```
[9, 2, 6, 3, 8, 5, 1]
```

1. 첫 번째 값: 9
2. 중간 값: 3 (인덱스 3)
3. 마지막 값: 1

이 세 값은 `[9, 3, 1]` 입니다. 이 값들 중 중간값은 `3` 이므로, `3` 이 피벗으로 선택

6.5 기수 정렬(Radix Sort)

- 비교 기반 정렬 알고리즘과 달리 원소 간의 직접적인 비교를 하지 않고 **자릿수**를 기준으로 정렬을 수행
- 정수나 문자열 같은 구조화된 데이터를 정렬할 때 유용
- 시간 복잡도가 $O(nk)$로 비교적 효율적:
  - n : 데이터의 개수
  - k: 데이터 중 가장 큰 값의 자릿수



6.5 기수 정렬(Radix Sort)

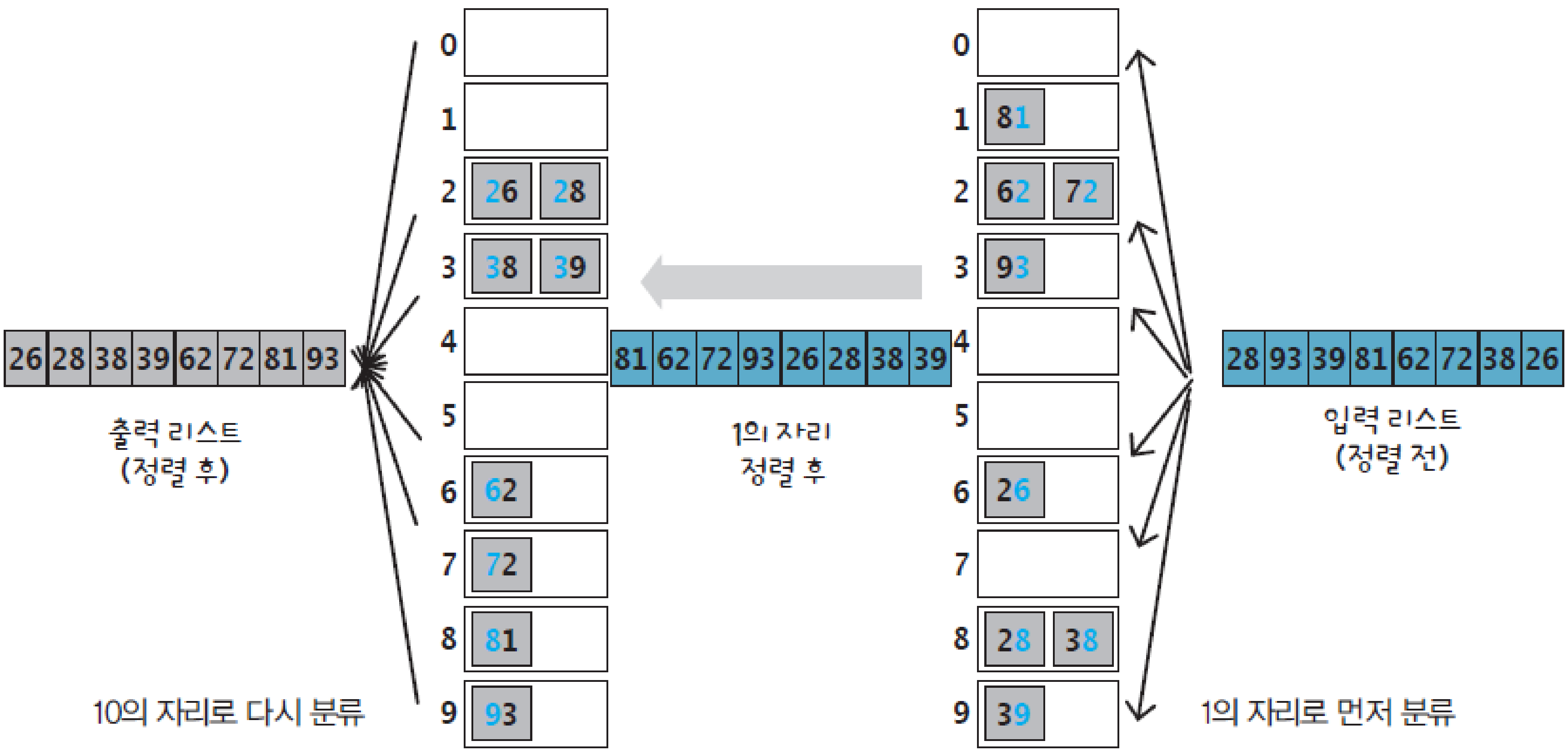
- 아이디어 → 배분
- 정렬해야 하는 값의 자릿수를 기준으로 차례대로 정렬을 수행
- **LSD(Least Significant Digit)**: **가장 낮은 자릿수부터** 정렬하는 방식
  – 보통 숫자 데이터에 사용되며, 각 자릿수를 기준으로 순차적으로 처리
- **MSD(Most Significant Digit)**: **가장 높은 자릿수부터** 정렬하는 방식
  – 주로 문자열과 같은 데이터에 사용

[170, 45, 75, 90, 802, 24, 2, 66]

1. **가장 낮은 자릿수(1의 자리) 기준으로 정렬**: [170, 90, 802, 2, 24, 45, 75, 66]
   - (0, 0, 2, 2, 4, 5, 5, 6 순으로 정렬)
2. **10의 자릿수 기준으로 정렬**: [802, 2, 24, 45, 66, 170, 75, 90]
   (0, 없음, 2, 4, 6, 7, 7, 9 순으로 정렬)
3. **100의 자릿수 기준으로 정렬**:[2, 24, 45, 66, 75, 90, 170, 802]
   - (없음, 없음, 없음, 없음, 없음, 없음, 1, 8 순으로 정렬)
4. **최종 결과** : [2, 24, 45, 66, 75, 90, 170, 802]

예: 자연수의 정렬

입력 데이터: [28, 93, 39, 81, 62, 72, 38, 26]			
1의 자리로 정렬	[81, 62, 72, 93, 26, 28, 38, 39]	10의 자리로 정렬	[28, 26, 39, 38, 62, 72, 81, 93]
10의 자리로 정렬	[26, 28, 38, 39, 62, 72, 81, 93] 정렬 성공	1의 자리로 정렬	~~[81, 62, 72, 93, 26, 28, 38, 39]~~ 정렬 실패



원형 큐를 사용한 기수 정렬 알고리즘

- 원형 큐 이용 : 고정된 크기를 가진 큐이며, 배열을 재사용하기 때문에 메모리를 절약

```python
class CircularQueue:
    def __init__(self, size):
        self.queue = [None] * size  # 고정된 크기의 리스트
        self.max_size = size
        self.front = -1
        self.rear = -1

    def is_full(self):
        return (self.rear + 1) % self.max_size == self.front

    def is_empty(self):
        return self.front == -1

    def enqueue(self, value):
        if self.is_full():
            raise Exception("Queue is full")
        if self.front == -1:
            self.front = 0
        self.rear = (self.rear + 1) % self.max_size
        self.queue[self.rear] = value

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        value = self.queue[self.front]
        if self.front == self.rear:  # 큐에 하나의 원소만 있을 때
            self.front = -1
            self.rear = -1
        else:
            self.front = (self.front + 1 self.max_size
        return value
```

```python
# 기수 정렬 함수 수정
def radix_sort(A):
    BUCKETS = 10
    DIGITS = 3  # 정렬할 숫자의 자릿수, 예를 들어 3자리까지

    # BUCKETS개의 원형 큐 생성
    queues = []
    for _ in range(BUCKETS):
        queues.append(CircularQueue(len(A)))

    n = len(A)
    factor = 1

    for d in range(DIGITS):
        # 각 자릿수에 대해 정렬
        for i in range(n):
            digit = (A[i] // factor) % BUCKETS
            queues[digit].enqueue(A[i])  # 자릿수에 해당하는 큐에 삽입

        i = 0
        # 큐에서 다시 배열로 값을 꺼내어 정렬된 상태로 재배열
        for b in range(BUCKETS):
            while not queues[b].is_empty():
                A[i] = queues[b].dequeue()
                i += 1

        factor *= BUCKETS  # 다음 자릿수로 이동

    # 처리 과정 출력 (선택 사항)
    print("정렬 결과:", A)
```

테스트 프로그램

```python
import random            # 난수 발생을 위해 random 모듈 포함
BUCKETS = 10             # 10진법 사용
DIGITS  = 4              # 최대 4자릿수 숫자를 정렬함

# 리스트 내포(list comprehension)로 난수 10개로 이루어진 리스트 생성
data = [random.randint(1,9999) for _ in range(10)]
radix_sort(data)
print("Radix:", data)
```

실행 결과

```
step 1 [941, 1233, 1554, 1314, 7044, 7944, 1165, 4376, 2587, 6059]
step 2 [1314, 1233, 941, 7044, 7944, 1554, 6059, 1165, 4376, 2587]
step 3 [7044, 6059, 1165, 1233, 1314, 4376, 1554, 2587, 941, 7944]
step 4 [941, 1165, 1233, 1314, 1554, 2587, 4376, 6059, 7044, 7944]
Radix: [941, 1165, 1233, 1314, 1554, 2587, 4376, 6059, 7044, 7944]
```

일, 십, 백, 천의 자리 순으로 정렬

최종 정렬 결과

기수 정렬의 특징

- 버킷(큐)의 개수는 키의 표현 방법과 밀접한 관계
  - 이진법을 사용한다면 버킷은 2개
  - 알파벳 문자를 사용한다면 버킷은 26개
  - 십진법을 사용한다면 버킷은 10개
- n개의 레코드, d개의 자릿수 키의 기수 정렬
  - 메인 루프는 자릿수 d번 반복
  - 큐에 n개 레코드 입력 수행
- 시간 복잡도: $O(dn)$, 대부분 $d < 10$ 이하
- **비교 기반이 아님**: 원소 간 비교가 아니라 각 자릿수의 값을 기준으로 정렬하기 때문에 특정 상황에서는 비교 기반 정렬 알고리즘보다 더 효율적
- **대용량 데이터 처리 가능**: 특히 정수 데이터나 고정된 길이의 문자열 같은 경우에는 매우 빠르게 정렬을 수행
- **데이터 제약**: 기수 정렬은 숫자나 문자열처럼 자릿수로 구분할 수 있는 데이터에만 적용 - 실수, 한글, 한자로 이루어진 키는 정렬 못함
- **메모리 사용**: 내부적으로 기수 정렬은 추가적인 공간을 필요로 할 수 있기 때문에 메모리 사용이 많아짐
- **기본 정렬 알고리즘 필요**: 기수 정렬 자체는 각 자릿수를 정렬할 때 또 다른 안정적인 정렬 알고리즘이 필요- 보통 **계수 정렬(Counting Sort)**을 사용

계수 정렬(Counting Sort)

- 각 항목의 빈도를 계산하고, 그 빈도를 기반으로 원소들을 정렬
- **누적 카운트 배열**을 이용해 각 숫자가 **정렬된 배열에서 차지할 위치**를 계산
- 특히 **정수 범위**가 제한된 경우에 매우 효율적
- 시간 복잡도는 $O(n + k)$, n : 데이터의 개수, k: 가장 큰 값의 자릿수
- **계수 정렬의 동작 방식**
  1. **입력 배열의 최대값 찾기**: 입력 배열에서 가장 큰 정수를 구함.
  2. **카운트 배열 생성**: 0부터 최대값까지의 범위를 가지는 카운트 배열을 생성. 각 숫자가 배열에 몇 번 등장 했는지를 저장.
  3. **카운트 배열 업데이트**: 입력 배열의 각 숫자를 세고, 그 빈도를 카운트 배열에 저장.
  4. **누적합 계산**: 카운트 배열에서 이전 값들과 더해지도록 누적합을 계산하여, 각 숫자가 배열에서 차지할 최종 위치를 찾는다.
  5. **출력 배열 생성**: 입력 배열을 역순으로 처리하면서, 누적 카운트 배열을 이용해 정렬된 위치에 숫자를 배치한다.

예: 계수 정렬(Counting Sort)

1. 입력 배열 : arr = [4, 2, 2, 8, 3, 3, 1]
2. 카운트 배열 생성: [0, 1, 2, 2, 1, 0, 0, 0, 1]
   - 입력 배열에서 가장 큰 값은 8이다. 따라서 0부터 8까지 인덱스를 가지는 카운트 배열을 만들고, 각 숫자가 배열에 몇 번 등장하는지를 기록
3. 누적 카운트 배열: [0, 1, 3, 5, 6, 6, 6, 6, 7]
   - 카운트 배열을 사용하여 누적 카운트 배열을 계산함. 누적 카운트는 이전 숫자들까지의 빈도수를 더해 각 숫자가 정렬된 배열에서 차지하는 마지막 위치를 알 수 있게 함
   - 이 배열의 값은 각 숫자가 정렬된 배열에서 차지할 **마지막** 위치를 표시

숫자	0	1	2	3	4	5	6	7	8
빈도	0	1	2	2	1	0	0	0	1

숫자	0	1	2	3	4	5	6	7	8
누적	0	1	3	5	6	6	6	6	7

예: 계수 정렬(Counting Sort)

4. 출력 배열: [1, 2, 2, 3, 3, 4, 8]
   - 이제 입력 배열을 **역순으로 처리**하면서, 누적 카운트 배열을 이용해 각 숫자를 정렬된 위치에 배치
   - 역순으로 처리하는 이유는 **안정성**을 보장하기 위함
     - 같은 값을 가진 요소들 사이의 원래 순서를 유지

- 각 숫자가 배열에서 차지할 위치는 누적 카운트 배열에 의해 결정되며, 입력 배열을 역순으로 처리하면서 정렬된 배열에 차례대로 배치

숫자	입력 배열	출력 배열	설명
1	[4, 2, 2, 8, 3, 3, 1]	[1, , , , , ,]	- 누적 카운트 배열에서 1은 1번째 위치 - 누적 배열의 1의 값을 1에서 0으로 감소
3	[4, 2, 2, 8, 3, 3, 1]	[1, , , , 3 , ,]	- 누적 카운트 배열에서 3은 5번째 위치 - 누적 배열의 3의 값을 5에서 4으로 감소
3	[4, 2, 2, 8, 3, 3, 1]	[1, , , 3 , 3 , ,]	- 누적 카운트 배열에서 3은 4번째 위치 - 누적 배열의 3의 값을 4에서 3으로 감소
8	[4, 2, 2, 8, 3, 3, 1]	[1, , , 3 , 3 , , 8]	- 누적 카운트 배열에서 8은 7번째 위치 - 누적 배열의 8의 값을 7에서 6으로 감소
2	[4, 2, 2, 8, 3, 3, 1]	[1, , 2 , 3 , 3 , , 8]	- 누적 카운트 배열에서 2는 3번째 위치 - 누적 배열의 2의 값을 3에서 2으로 감소
2	[4, 2, 2, 8, 3, 3, 1]	[1, 2 , 2 , 3 , 3 , , 8]	- 누적 카운트 배열에서 2는 2번째 위치 - 누적 배열의 2의 값을 2에서 1으로 감소
4	[4, 2, 2, 8, 3, 3, 1]	[1, 2 , 2 , 3 , 3 ,4 , 8]	- 누적 카운트 배열에서 4는 6번째 위치 - 누적 배열의 4의 값을 6에서 5으로 감소

6.6 파이썬의 정렬함수 활용하기

- 리스트의 sort( ) 메서드

```
data = [6,3,7,4,9,1,5,2,8]
data.sort()                   # data: [1,2,3,4,5,6,7,8,9]

data = [6,3,7,4,9,1,5,2,8]
data.sort(reverse=True)       # data: [9,8,7,6,5,4,3,2,1]
```

- 파이썬의 내장 함수 sorted( )

```
data = [6,3,7,4,9,1,5,2,8]
result = sorted(data)         # result: [1,2,3,4,5,6,7,8,9]
                              # data : [6,3,7,4,9,1,5,2,8]

result = sorted(data, reverse=True) # result: [9,8,7,6,5,4,3,2,1]
```

복잡한 레코드의 정렬은 어떻게 할까요?

- (예) 3차원 공간상의 점들의 리스트 정렬

```
data = [(62, 88, 81), (50, 3, 31), (86, 53, 42), (73, 47, 4), (89, 9, 8),
(47, 88, 55), (19, 18, 20), (15, 1, 88), (90, 6, 60), (41, 92, 19)]
```

- 키워드 인수 key
  - x값의 오름차순으로 정렬 예

```
def keyfunc( p ):         # 레코드 p에서 키를 반환하는 함수. p=(x,y,z)
    return p[0]           # p의 첫 번째 요소(p[0], x값)를 키로 반환

print("data  :", data)
x_inc = sorted(data, key = keyfunc)
print("x_inc :", x_inc )
```

실행 결과

```
data  : [(62, 88, 81), (50, 3, 31), (86, 53, 42), (73, 47, 4), (89, 9,
8), (47, 88, 55), (19, 18, 20), (15, 1, 88), (90, 6, 60), (41, 92, 19)]
x_inc : [(15, 1, 88), (19, 18, 20), (41, 92, 19), (47, 88, 55), (50, 3,
31), (62, 88, 81), (73, 47, 4), (86, 53, 42), (89, 9, 8), (90, 6, 60)]
```

람다 함수를 이용한 키 지정

- y값의 내림차순으로 정렬 예

```
y_dec = sorted(data, key = lambda p : p[1], reverse=True)
print("data  :", data)
print("y_dec :", y_dec)
```

실행 결과

```
data  : [(62, 88, 81), (50, 3, 31), (86, 53, 42), (73, 47, 4), (89, 9,
8), (47, 88, 55), (19, 18, 20), (15, 1, 88), (90, 6, 60), (41, 92, 19)]
y_dec : [41, 92, 19), (62, 88, 81), (47, 88, 55), (86, 53, 42), (73, 47,
4), (19, 18, 20), (89, 9, 8), (90, 6, 60), (50, 3, 31), (15, 1, 88)]
```

람다 함수를 이용한 키 지정

- 크기의 오름차순으로 정렬 예 크기는 $\sqrt{x^2 + y^2 + z^2}$.

```
import math
magni = sorted(data, key = lambda p : math.sqrt(p[0]*p[0]+p[1]*p[1]+p[2]*p[2]))
print("data  :", data)
print("magni :", magni)
```

실행 결과

```
data  : [(62, 88, 81), (50, 3, 31), (86, 53, 42), (73, 47, 4), (89, 9, 8),
(47, 88, 55), (19, 18, 20), (15, 1, 88), (90, 6, 60), (41, 92, 19)]
magni : [(19, 18, 20), (50, 3, 31), (73, 47, 4), (15, 1, 88), (89, 9, 8),
(41, 92, 19), (90, 6, 60), (86, 53, 42), (47, 88, 55), (62, 88, 81)]
```