가천대학교
Gachon University

04

Algorithm

정렬 알고리즘1

2025-03-28

조윤실

목 차

- 정렬 알고리즘

1) 정렬 알고리즘 개요
2) 기초 정렬 알고리즘
   - 버블 정렬, 선택 정렬, 삽입 정렬
3) 고급 정렬 알고리즘
   - 병합 정렬, 퀵 정렬, 힙 정렬, 셸 정렬
4) 특수 정렬 알고리즘
   - 계수 정렬, 기수 정렬, 버킷 정렬

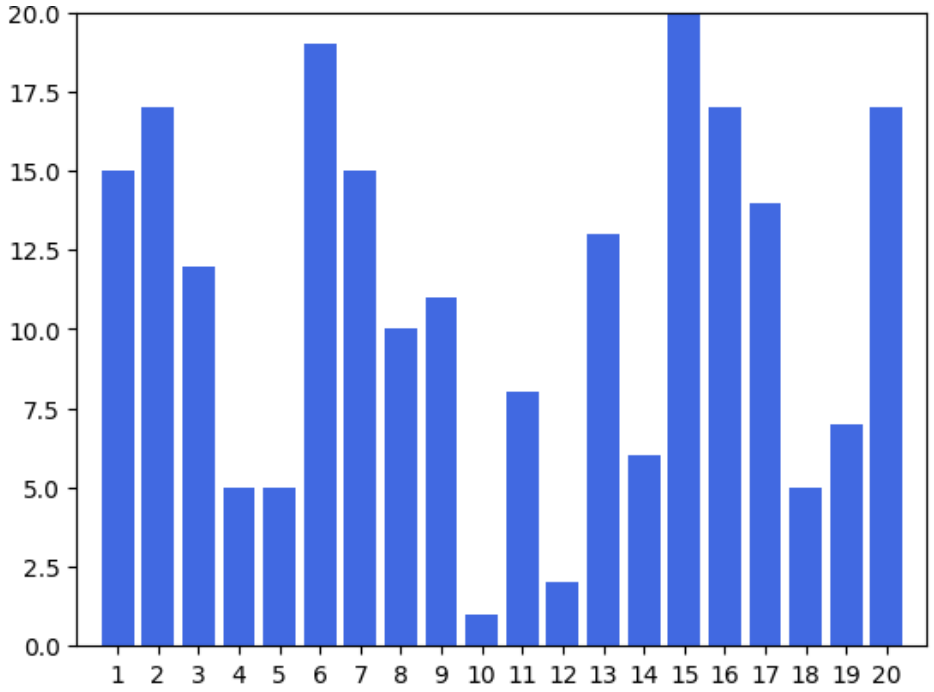
※ 가천대학교 컴퓨터공학과 '알고리즘' 과정

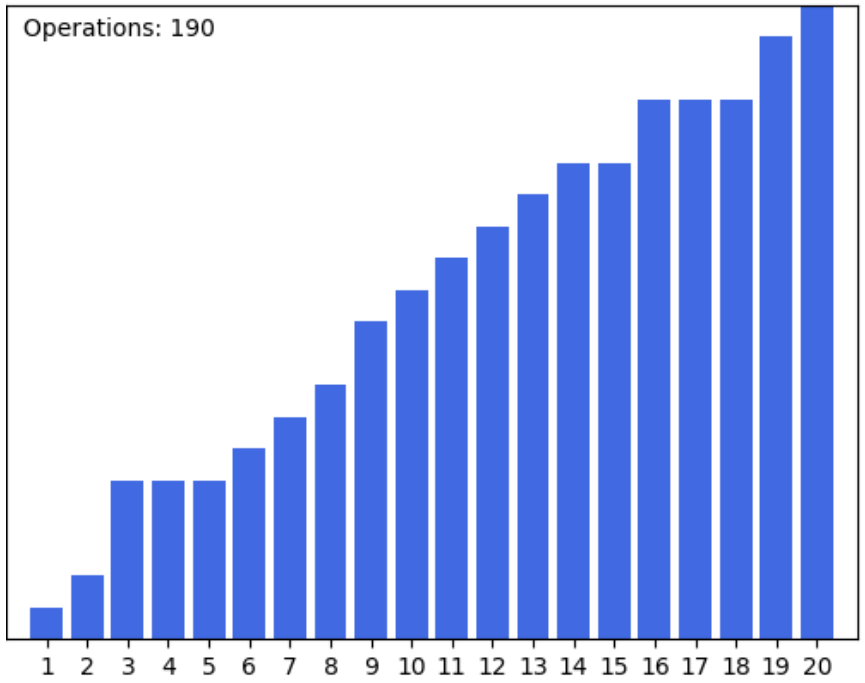
정렬 알고리즘1

정렬 알고리즘

[Quiz]

- 임의의 정수 20개를 오름 차순으로 정렬하는 파이썬 코드를 작성해 보세요.





정렬 알고리즘이란?

- 정렬 알고리즘(Sorting Algorithm)
  - 정렬은 원소들을 순서대로 배열하는 것(ascending order/ descending order)
  - 정렬은 그 자체로 매우 중요한 주제이기도 하고, 알고리즘의 설계와 분석,

    생각하는 방법 등을 훈련하기에 좋은 구성 요소를 많이 가지고 있음.
  - 알고리즘 분야에서 사용하는 여러 핵심 기술이 정렬에 포함되어 있어

    정렬을 잘 이해하면 다른 주제를 이해하는데 큰 도움이 됨

정렬 알고리즘과 컴퓨터 과학 분야의 핵심 기술

- 정렬 알고리즘에 포함되어 있는 알고리즘 여러 핵심 기술
  - 재귀(Recursion) : 자기 자신을 다시 호출하여 문제를 작은 단위로 분할하고 해결
  - 분할 정복(Divide and Conquer) : 큰 문제를 작은 문제로 분할하고, 작은 문제를 해결한 뒤 결합
  - 탐색(Searching) : 데이터 집합에서 특정 값을 찾는 알고리즘
  - 자료 구조(Data Structures) : 정렬을 위한 효율적인 데이터 저장 및 조작
  - 병렬 처리(Parallel Processing) : 작업을 여러 쓰레드로 분할하여 병렬 처리
  - 동적 프로그래밍(Dynamic Programming) : 이전 계산 결과를 재활용하여 효율성 증대
  - 확률 및 통계(Probability and Statistics) : 입력 데이터의 통계적 특성을 이용

주요 정렬 알고리즘의 연표

년대	주요 알고리즘	특징	시간 복잡도(AVG)
~ 1950년대	Bubble Sort(? John McCarthy)	- 직관적이고 간단하지만, 비효율적	$O(n^2)$
	Insertion Sort(John von Neumann)	- 작은 데이터 삽입에 효율적, 간단한 구현	$O(n^2)$
	Selection Sort(Robert Sedgewick)	- 버블보다 효율적, 최악 경우 시간 복잡도가 높음	$O(n^2)$
	Merge Sort(John von Neumann)	- 데이터를 분할하고 정렬된 것을 병합하는 방식	$O(n \log n)$
	Heap Sort(J. W. J. Williams)	- 완전 이진 트리를 활용, 비교적 빠른 속도	$O(n \log n)$
	Radix Sort(Herman Hollerith)	- 계수 정렬과 유사, 숫자의 자릿수를 기준으로 정렬	$O(kn)$
~ 1990년대	**Quick Sort**(Tony Hoare)	- 가장 빠른 정렬 알고리즘 중 하나	$O(n \log n)$
	Shell Sort(Donald Shell)	- 삽입 정렬의 개선 버전, data크기에 따라 달라짐	$O(n \log n)$
	Counting Sort(Harold H. Seward)	- 특정 범위의 정수 데이터에 효율적, O(n + k) 시간 복잡도 (k는 데이터 범위)	$O(n + k)$
2000년대 ~	Tim Sort(Tim Peters) Burst Sort(Robert Sedgewick) PDQsort(Walter Bright) ---	- 병합 정렬과 삽입 정렬의 기법을 결합하여 최적화한 알고리즘, 하이브리드 - 퀵 정렬, 힙 정렬, 삽입 정렬 등의 하이브리드	$O(n \log n)$

[Quiz]

- 파이썬 정렬 함수에 적용된 파이썬 알고리즘은?

정렬 알고리즘 분류 기준

- 시간 복잡도에 따른 분류
- 정렬 방식에 따른 분류
- 안정성에 따른 분류
- 메모리 사용에 따른 분류
- 정렬 과정에 따른 분류

정렬 알고리즘 분류 기준

- **시간 복잡도**에 따른 분류
  - O(n^2) 복잡도 : Bubble Sort, Selection Sort, Insertion Sort 등
  - O($n \log n$) 복잡도 : Quick Sort, Merge Sort, Heap Sort 등
  - O(n)(선형) 복잡도 : Counting Sort, Radix Sort 등

정렬 알고리즘 분류 기준

- **정렬 방식**에 따른 분류
  - 비교 정렬(Comparison Sort) : 데이터 간 비교를 통해 정렬

    Bubble Sort, Quick Sort, Merge Sort 등
  - 분포 정렬(Distribution Sort) : 데이터 값의 분포를 이용하여 정렬

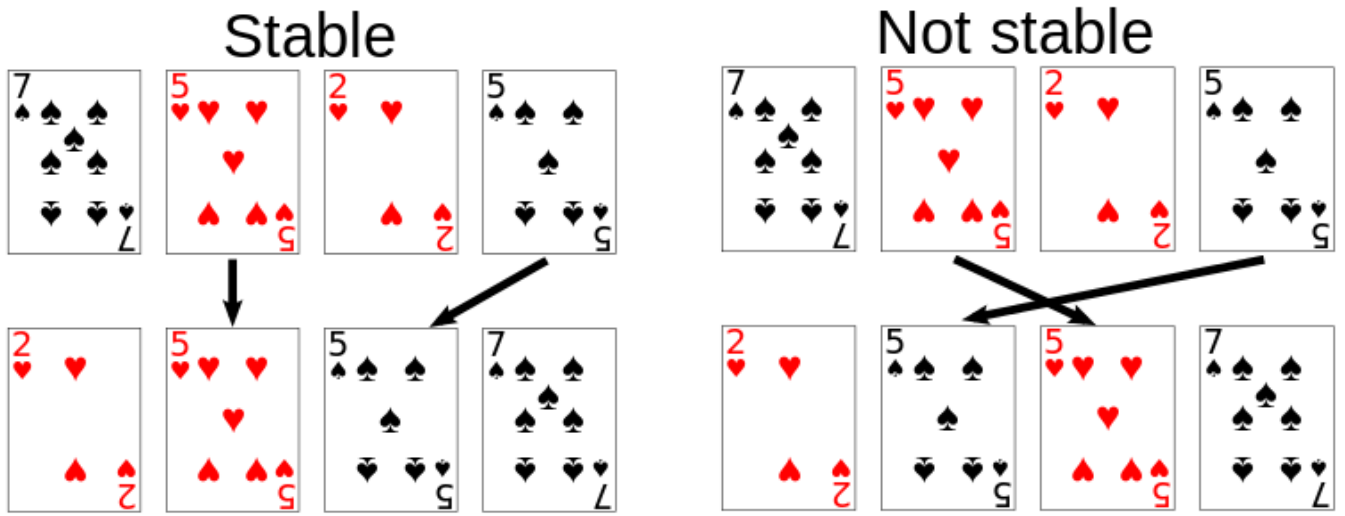
    Counting Sort, Radix Sort 등

정렬 알고리즘 분류 기준

- **안정성**에 따른 분류
  - 안정 정렬(Stable Sort): 동일한 값을 가진 원소의 상대적 위치가 바뀌지 않음

    Merge Sort, Insertion Sort 등
  - 불안정 정렬(Unstable Sort): 동일한 값을 가진 원소의 상대적 위치가 바뀔 수 있음

    Quick Sort, Shell Sort 등



이미지 출처: https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/

※본 강의 자료는 본인 학습용으로만 사용 가능하며 무단 복제/배포를 금지합니다.

정렬 알고리즘 분류 기준

- **메모리 사용에 따른 분류**
  - 제자리 정렬(In-place Sort): 추가 메모리 사용 없이 입력 배열 내에서 정렬
    공간 복잡도
    Quick Sort, Shell Sort 등
  - 보조 메모리 정렬(Auxiliary-Space Sort): 추가 메모리를 사용하여 정렬
    Merge Sort 등
- **정렬 과정에 따른 분류**
  - 내부 정렬(Internal Sort): 정렬할 데이터가 메모리 내에 모두 존재
  - 외부 정렬(External Sort): 정렬할 데이터가 너무 커서 외부 저장 장치를 사용
    Merge Sort 기반

정렬 알고리즘 복잡도

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

이미지 출처: https://lamfo-unb.github.io/2019/04/21/Sorting-algorithms/

기초 정렬 알고리즘

기초 정렬 알고리즘

- 기초 정렬 알고리즘
  - 평균적으로 O(n^2)의 시간이 소요되는 정렬 알고리즘들
    - Bubble Sort(버블 정렬)
    - Selection Sort(선택 정렬)
    - Insertion Sort(삽입 정렬)
  - 이 알고리즘들은 간단한 구현과 이해의 용이성 때문에 교육 목적으로 자주 사용됨.
  - 데이터의 양이 많아질수록 성능이 급격히 저하됨

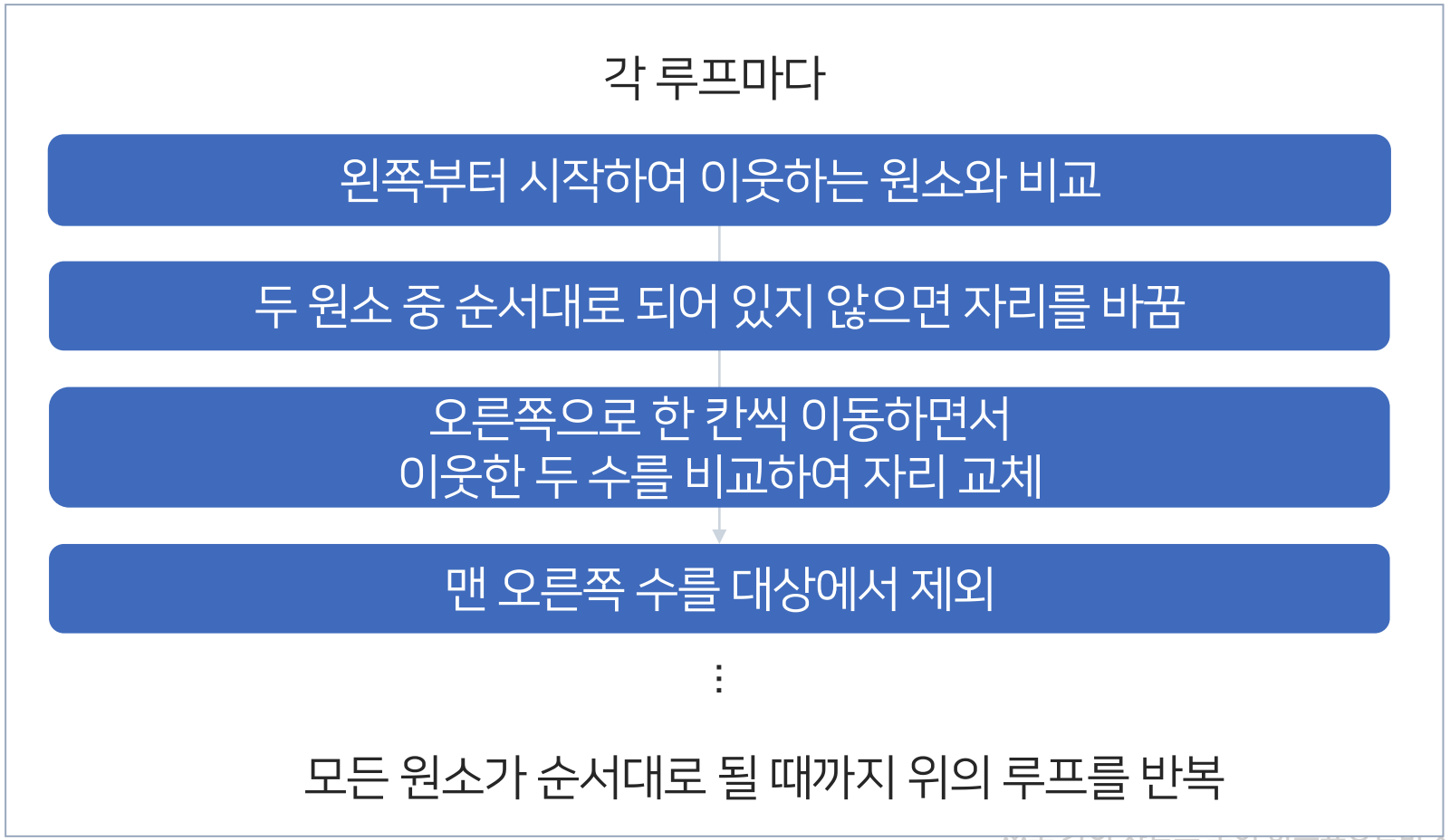
버블 정렬(Bubble Sort)

배열을 반복적으로 순회하면서 인접한 두 요소를 비교하고, 제일 큰 원소를 배열의 맨 뒤(또는 앞)으로 옮기는 방식으로 동작하는 정렬 알고리즘

- 배열을 반복하면서 더 큰(또는 작은) 원소가 "거품처럼" 위로 올라가는 모습과 비슷
- 버블 정렬은 구현이 간단하고 이해하기 쉽지만, 정렬할 배열의 길이가 길어질수록 비효율적인 알고리즘
- 데이터가 거의 정렬되어 있는 경우 조기 종료를 통해 $O(n)$으로 개선 가능
- 최선/최악 시간 복잡도: $O(n^2)$

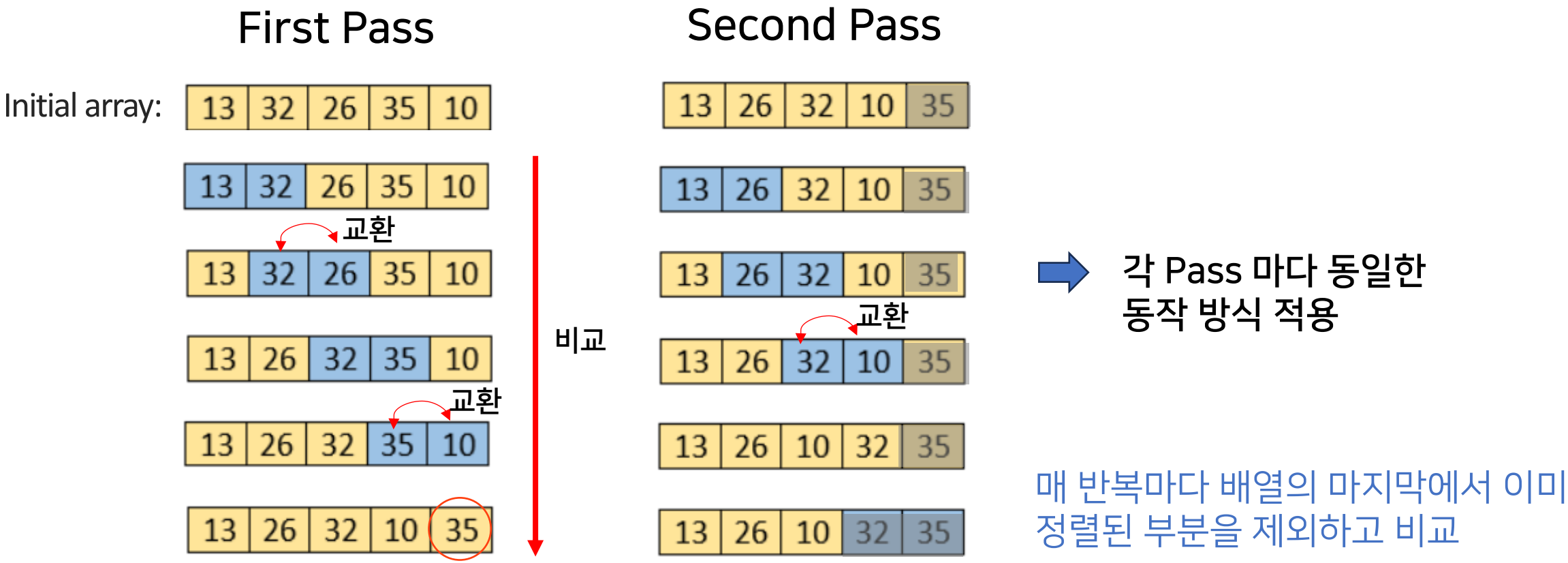
버블 정렬(Bubble Sort)

- 버블 정렬 동작 방식



버블 정렬(Bubble Sort)

- 버블 정렬 동작 방식



각 Pass 마다 동일한
동작 방식 적용

매 반복마다 배열의 마지막에서 이미
정렬된 부분을 제외하고 비교

버블 정렬(Bubble Sort)

▪ 버블 정렬 알고리즘

```
bubbleSort(A[ ], n):                ▷ A[0...n-1]을 정렬한다.
    ❶ for last ← n-1 downto 1
        ❷ for i ←  0 to last-1
            ❸ if (A[i] > A[i+1])
                A[i] ↔ A[i+1]  ▷ 원소 교환
```

[수행 시간]

- ❶의 for 루프는 n-1 번 반복
- ❷의 for 루프는 각각 n-1, n-2, ..., 2, 1 번 반복
- ❸의 상수 시간 작업

$$(n-1)+(n-2)+\cdots+2+1 = \Theta(n^2)$$

실습 : 버블 정렬(Bubble Sort) 알고리즘 구현하기

- 버블 정렬 알고리즘 파이썬 코드로 구현하기

bubbleSort(*A[ ]*, *n*): ▷ *A*[0...*n*-1]을 정렬한다.

❶ for *last* ← *n-1* downto 1

❷ for *i* ← 0 to *last*-1

❸ if *(A[i] > A[i+1])*

A[i] ↔ *A[i+1]* ▷ 원소 교환

```
    return arr

# 예시 사용
arr = [13, 32, 26, 35, 10]
print("정렬된 배열:", bubble_sort(arr))
```

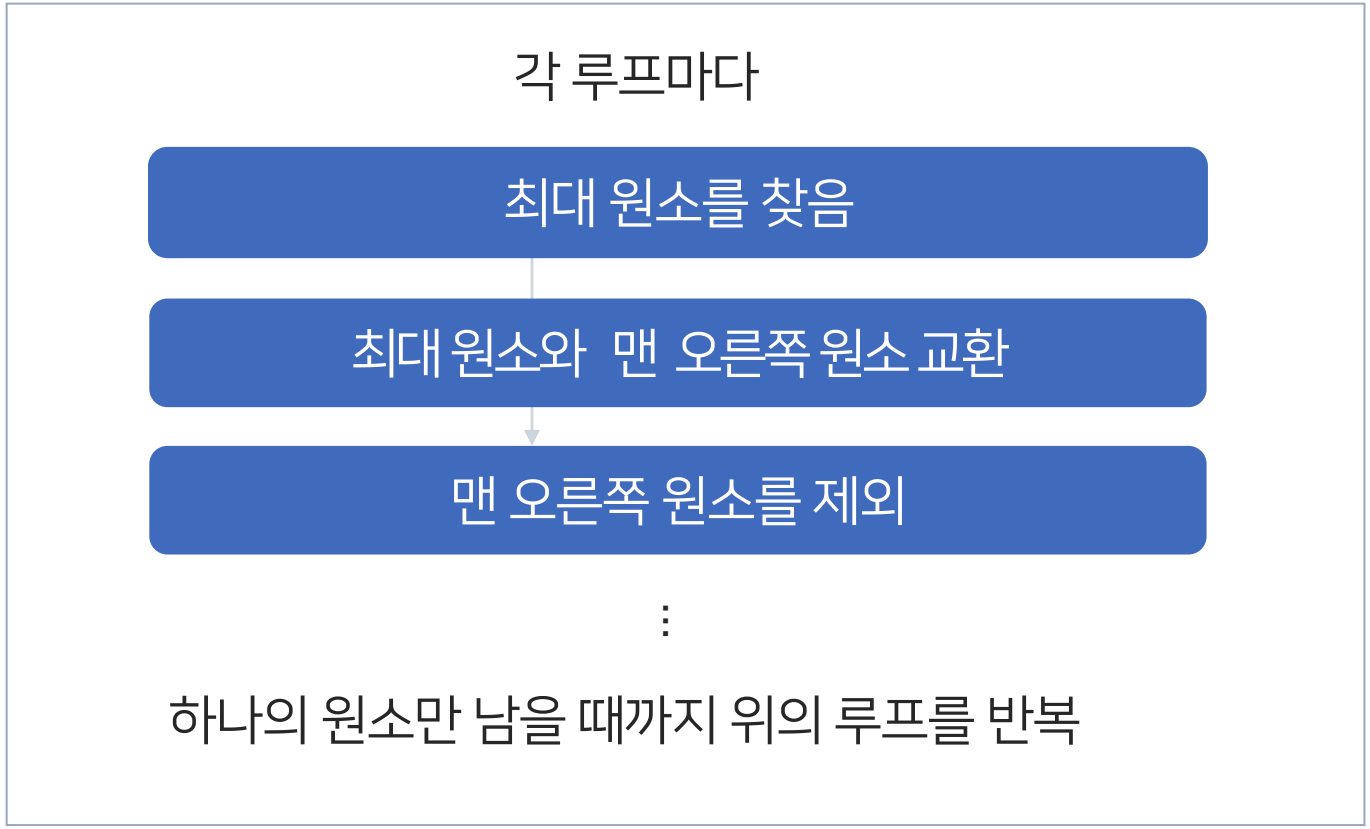
선택 정렬(Selection Sort)

배열을 반복하여 가장 큰(또는 가장 작은) 원소를 선택하여 해당 원소를
배열의 맨 뒤(또는 앞)으로 옮기는 방식으로 동작하는 정렬 알고리즘

- 데이터가 거의 정렬되어 있더라도 시간 복잡도가 변하지 않음
- 교환 횟수가 최소화되지만 비교 횟수는 많음
- 구현은 간단하지만, 다른 정렬 알고리즘에 비해 비효율적
- 정렬할 배열의 크기가 커질수록 성능이 저하
- 최선/최악 시간 복잡도: $O(n^2)$

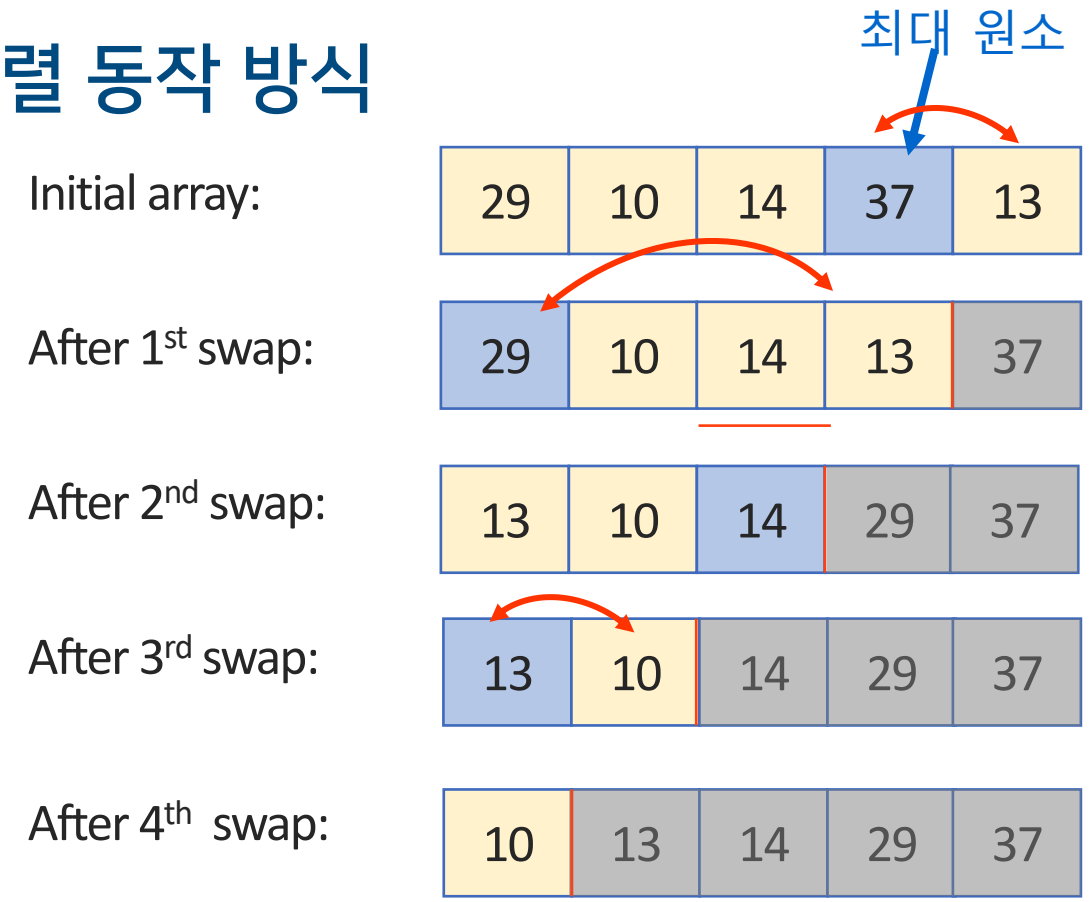
선택 정렬(Selection Sort)

- 선택 정렬 동작 방식



선택 정렬(Selection Sort)

- 선택 정렬 동작 방식



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

Worst case
Average case

선택 정렬(Selection Sort)

▪ 선택 정렬 알고리즘

selectionSort($A[\,]$, n): ▷ $A[0...n\text{-}1]$을 정렬한다.

❶ for $last \leftarrow n\text{-}1$ downto 1

❷ $A[0...last]$ 중 가장 큰 수 A[k]를 찾는다.

❸ $A[k] \leftrightarrow A[last]$ ▷ $A[k]$와 $A[last]$의 값 교환

[수행 시간]

- ❶의 for 루프는 n-1 번 반복
- ❷에서 가장 큰 수를 찾기 위한 비교 횟수 : n-1, n-2, ..., 2, 1
- ❸의 교환은 상수 시간 작업

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

실습 : 선택 정렬(Selection Sort) 알고리즘 구현하기

- 선택 정렬 알고리즘 파이썬 코드로 구현하기

selectionSort(*A[ ]*, *n*): ▷ *A*[0...*n*-1]을 정렬한다.

❶ for *last* ← *n-1* downto 1

❷ *A*[0...*last*] 중 가장 큰 수 A[*k*]를 찾는다.

❸ *A*[*k*] ↔ *A*[*last*] ▷ *A*[*k*]와 *A*[*last*]의 값 교환

```
    return arr

# 예시 사용
arr = [12, 29, 25, 8, 32, 17, 40]
print("정렬된 배열:", selection_sort(arr))
```

삽입 정렬(Insertion Sort)

배열을 정렬된 부분과 정렬되지 않은 부분으로 나누고, 정렬되지 않은 부분의 원소를 정렬된 부분에 삽입하는 방식으로 동작하는 정렬 알고리즘

- 간단하면서도 효율적인 정렬 알고리즘 중 하나
- 정렬할 배열이 이미 거의 정렬되어 있는 경우에는 효율적
- 데이터셋의 크기가 작을 때에도 성능이 좋음
- 최선 시간 복잡도: $O(n)$
- 최악 시간 복잡도: $O(n^2)$

삽입 정렬(Insertion Sort)

▪ 삽입 정렬 동작 방식

배열에서 첫 번째 요소는 이미 정렬된 것으로 간주, 두 번째 요소부터 시작

현재 요소(정렬되지 않은 부분의 첫 번째 요소)를
정렬된 부분의 원소와 차례로 비교하여 삽입할 적절한 위치를 찾음

↓

위치를 찾으면 정렬된 부분에서 이 위치 이후의 원소를
한 칸씩 이동시킴

↓

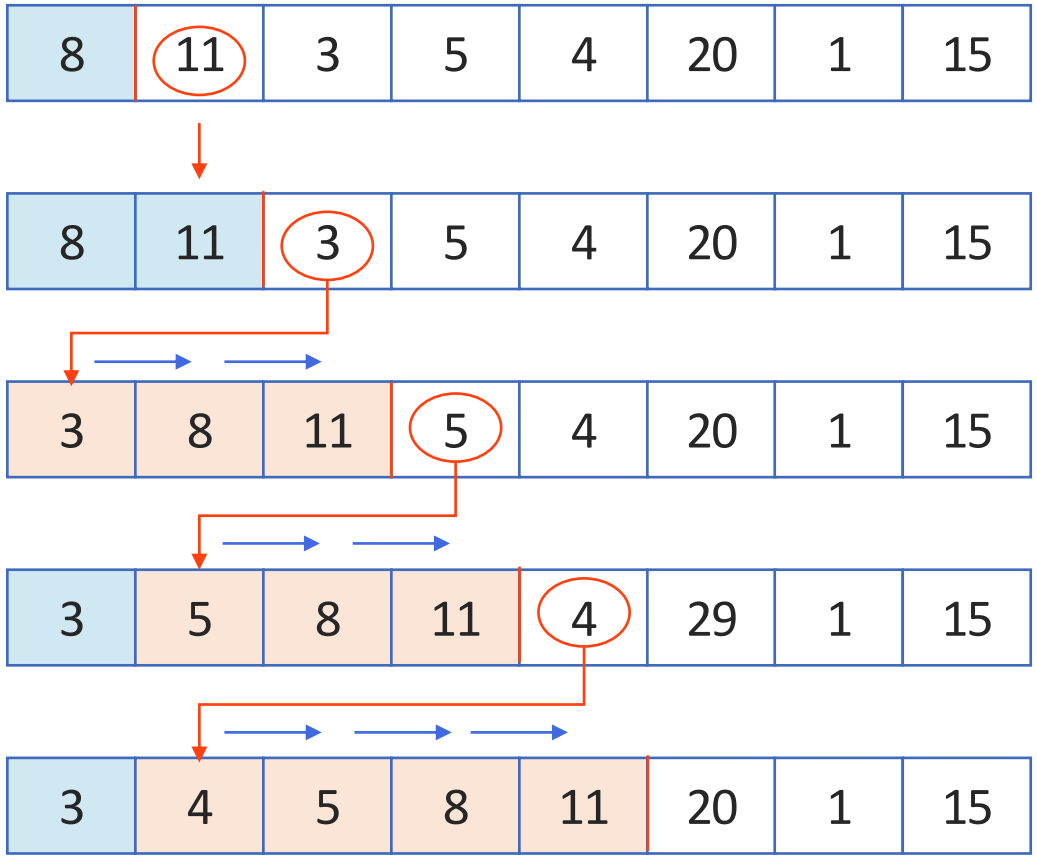
위치에 현재 원소를 삽입

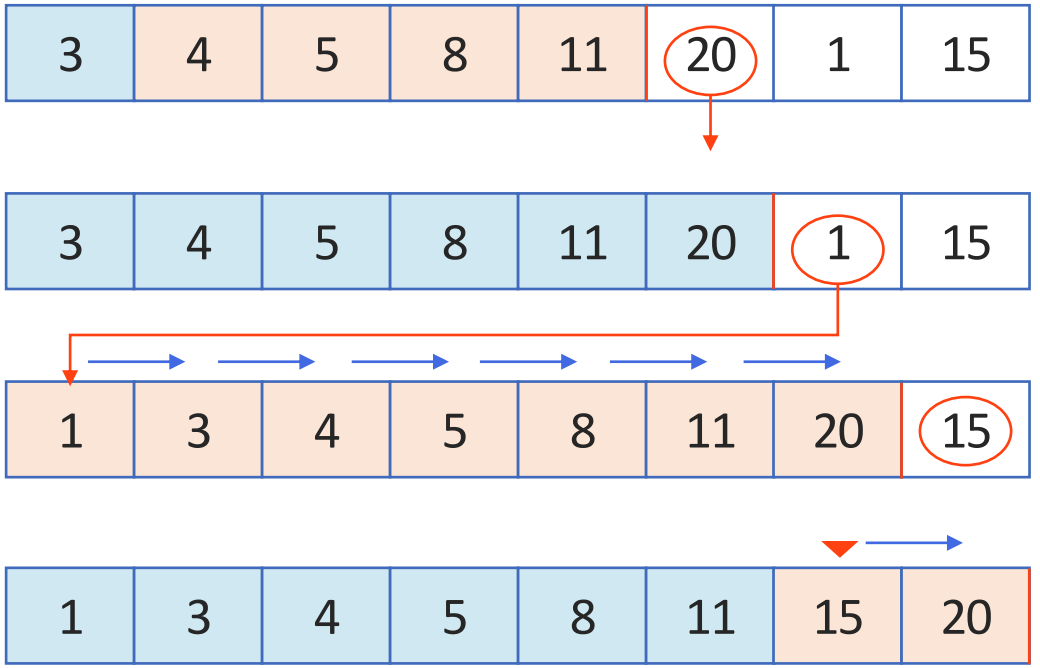
⋮

모든 원소가 순서대로 될 때까지 위의 루프를 반복

삽입 정렬(Insertion Sort)

- 삽입 정렬 동작 방식





수행 시간

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

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

삽입 정렬(Insertion Sort)

- 삽입 정렬 알고리즘

insertionSort(A[], n): ▷ A[0...n-1]을 정렬한다.

1. for $i \leftarrow 1$ to n-1
    2. A[0...i]의 적합한 자리에 A[i]를 삽입한다.

insertionSort(A[], n): ▷ A[0...n−1]을 정렬한다.

1. for $i \leftarrow 1$ to n -1

    $newItem \leftarrow A[i]$

    ▷ 이 지점에서 A[0...i −1]은 이미 정렬되어 있는 상태다.

2. for ($j \leftarrow i$ -1; $0 \le j$ and $newItem < A[j]$; j−−)

    $A[j+1] \leftarrow A[j]$

    $A[j+1] \leftarrow newItem$

삽입 정렬(Insertion Sort)

- 삽입 정렬 알고리즘(재귀)


```
insertionSort(A[], n):          ▷ A[0...n−1]을 정렬한다.
    ❶ if (n > 1)
        ❷ insertionSort(A[], n−1)
        ❸ A[0...n-1]의 적합한 자리에 A[n−1]을 삽입한다.
```

실습 : 삽입 정렬(Insertion Sort) 알고리즘 구현하기

- 삽입 정렬 알고리즘 파이썬 코드로 구현하기

insertionSort(A[], n): ▷ A[0...n−1]을 정렬한다.

❶ for $i \leftarrow 1$ to n -1

$newItem \leftarrow A[i]$

▷ 이 지점에서 A[0...i −1]은 이미 정렬되어 있는 상태다.

❷ for ($j \leftarrow i$ -1; $0 \leq j$ and $newItem < A[j]$; j−−)

$A[j+1] \leftarrow A[j]$

$A[j+1] \leftarrow newItem$

```
    return arr

# 예시 사용
arr = [8, 11, 3, 5, 4, 20, 1, 15]
print("정렬된 배열:", insertion_sort(arr))
```

고급 정렬 알고리즘

고급 정렬 알고리즘

- 고급 정렬 알고리즘
  - 평균적으로 $O(n \log n)$ 의 시간이 소요되는 정렬 알고리즘
    - 병합 정렬(Merge Sort)
    - 퀵 정렬(Quick Sort)
    - 힙 정렬(Heap Sort)
    - 셸 정렬(Shell Sort)
  - 이 범주에 속하는 알고리즘들은 대규모 데이터셋에 대해 좋은 성능을 보이며, 실제 응용 프로그램에서 널리 사용됨.
  - 특히, 퀵 정렬은 평균적인 경우 매우 효율이지만, 최악의 경우 $O(n^2)$ 까지 성능이 저하될 수 있다.

병합 정렬(Merge Sort)

주어진 배열을 반으로 나눈 후 각 부분의 배열을 재귀적으로 정렬하고, 정렬된 부분 배열을 다시 병합하여 정렬된 배열을 생성하는 정렬 알고리즘

- 분할 정복(divide and conquer) 알고리즘의 대표적인 예
- 시간 복잡도 $O(n \log n)$의 일관된 성능 보장
- 같은 값을 가진 원소의 순서를 유지하는 안정적인 정렬 알고리즘
- 일반적으로 정렬할 배열의 크기에 비례하여 병합 할 때 추가 메모리가 필요
- 최선/최악 시간 복잡도: $O(n \log n)$

병합 정렬(Merge Sort)

- ## 병합 정렬 동작 방식

① 배열을 반으로 나눈다. (배열의 중간 지점을 찾아서 배열을 두 개의 부분 배열로 나눈다.)

② 각 부분 배열에 대해 재귀적으로 병합 정렬을 수행한다.

(부분 배열의 크기가 1이 될 때까지 재귀 호출을 반복하여 부분 배열을 정렬)

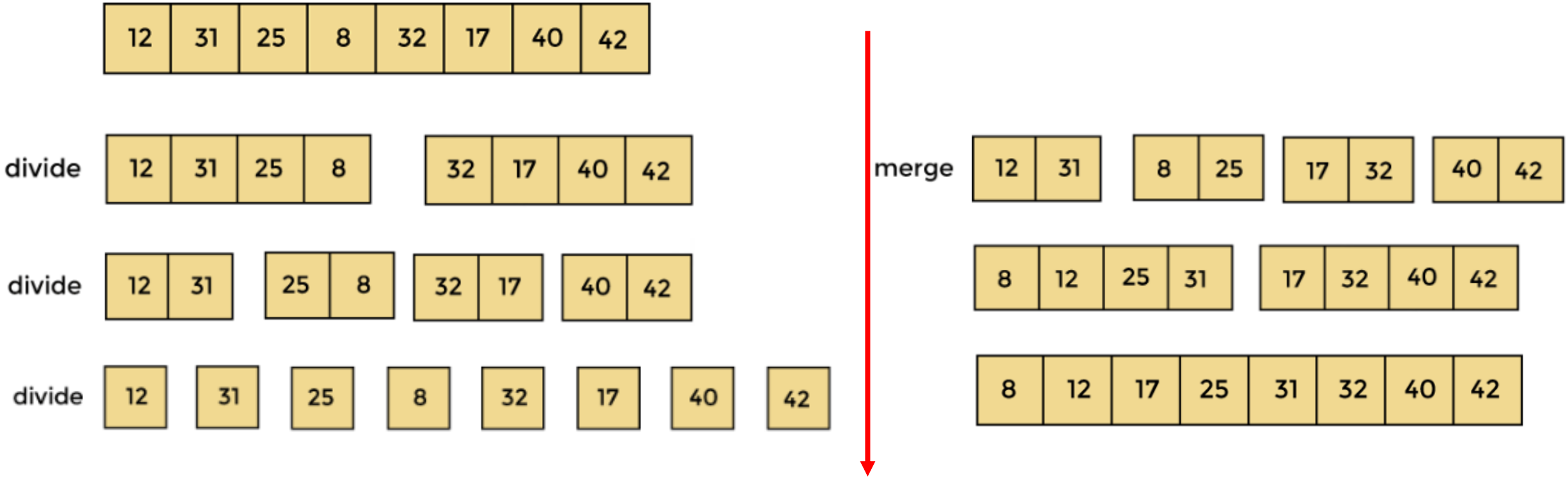
③ 정렬된 부분 배열을 병합(merge)하여 하나의 정렬된 배열을 생성

(이때, 두 부분 배열을 비교하면서 더 작은 값을 선택하여 새로운 배열에 추가)

④ 모든 부분 배열이 병합될 때까지 위의 과정을 반복

병합 정렬(Merge Sort)

- 병합 정렬 동작 방식



재귀 방법으로 분할

병합 정렬(Merge Sort)

- 병합 정렬 알고리즘

```
mergeSort(A[ ], p, r):        ▷ A[p...r]을 정렬한다.
    if (p < r)
        ❶ q ← ⌊(p+r)/2⌋          ▷ p, r의 중간 지점 계산
        ❷ mergeSort(A, p, q)     ▷ 전반부 정렬
        ❸ mergeSort(A, q+1, r)   ▷ 후반부 정렬
        ❹ merge(A, p, q, r)      ▷ 병합

merge(A[ ], p, q, r):
    정렬된 두 배열 A[p...q]와 A[q+1...r]을 합쳐
    정렬된 하나의 배열 A[p...r]을 만든다.
```

병합 정렬(Merge Sort)

- 병합 정렬 알고리즘

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # 배열을 반으로 나눔
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    # 각 부분 배열에 대해 재귀적으
    left = merge_sort(left)
    right = merge_sort(right)

    # 정렬된 부분 배열을 병합
    return merge(left, right)
```

```python
def merge(left, right):
    merged = []
    l_idx, r_idx = 0, 0

    # 두 부분 배열을 비교하면서 작은 값을 선택하여 병합
    while l_idx < len(left) and r_idx < len(right):
        if left[l_idx] < right[r_idx]:
            merged.append(left[l_idx])
            l_idx += 1
        else:
            merged.append(right[r_idx])
            r_idx += 1

    # 남은 요소들을 추가
    merged += left[l_idx:]
    merged += right[r_idx:]

    return merged
```

※본 정리 자료는 본인 학습용으로만 사용 가능하며 무단 복제/배포를 금지합니다.

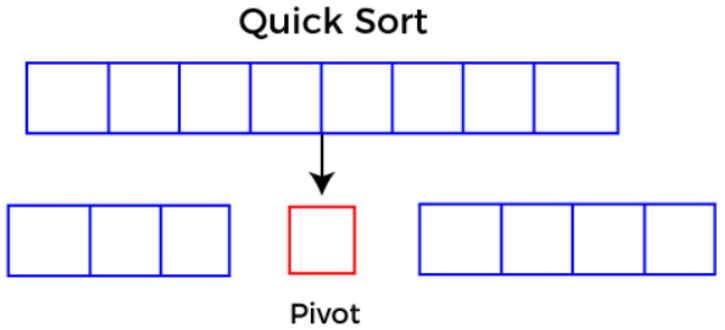
실습 : 병합 정렬(Merge Sort) 알고리즘 구현하기

- 병합 정렬 알고리즘 파이썬 코드로 구현하기

퀵 정렬(Quick Sort)

기준 원소를 하나 잡아 기준 원소보다 작은 원소와 큰 원소 그룹으로 나누어 기준 원소의 좌우로 분할한 다음 각각을 정렬하는 알고리즘

- 평균적으로 좋은 성능을 보여 현장에서 많이 쓰이는 알고리즘
- 분할 정복(divide and conquer) 방법을 사용하여 배열 정렬
- 배열을 피벗(pivot)을 기준으로 두 개의 부분 배열로 분할하고, 각 부분 배열을 재귀적으로 정렬하는 방식으로 동작
- 평균 시간 복잡도: $O(n \log n)$
- 최악 시간 복잡도: $O(n^2)$



퀵 정렬(Quick Sort)

- ## 퀵 정렬 동작 방식

① 배열에서 하나의 원소를 선택하여 피벗으로 설정

② 배열을 피벗을 기준으로 두 개의 부분 배열로 분할

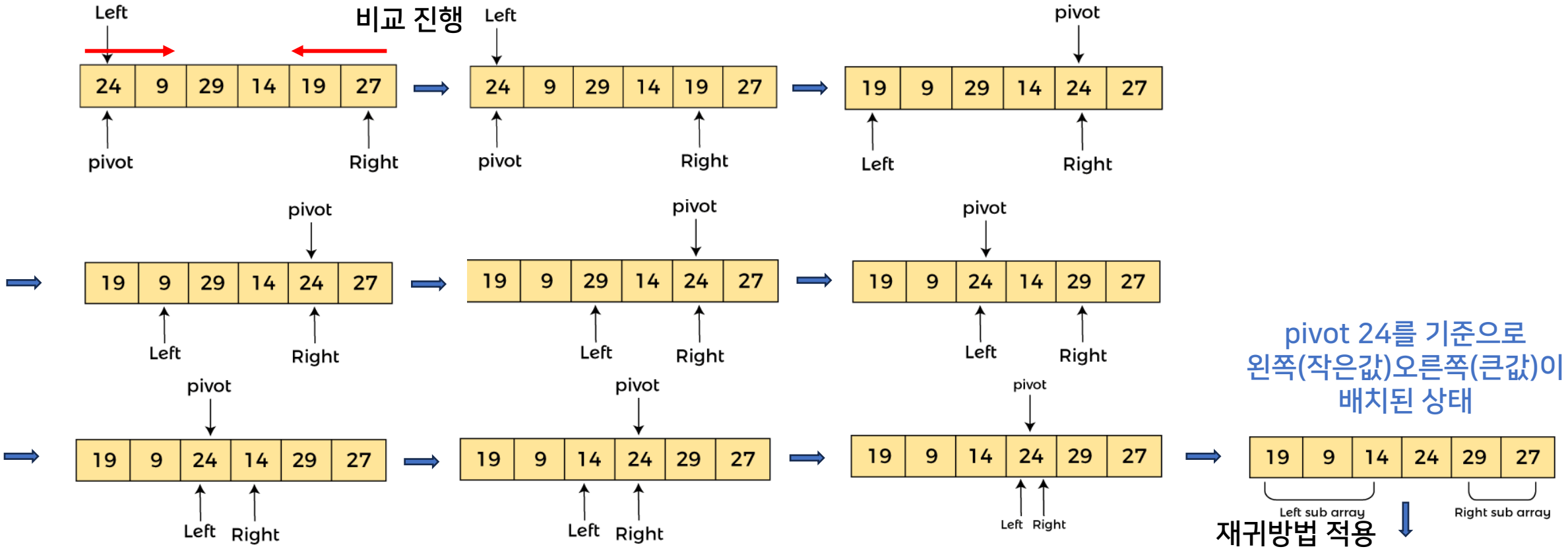
(왼쪽 부분 배열은 피벗보다 작은 원소들로, 오른쪽 부분 배열은 피벗보다 큰 원소들로 구성.
피벗은 이 단계에서 제 위치를 찾게 됨)

③ 왼쪽 부분 배열과 오른쪽 부분 배열에 대해 재귀적으로 위의 과정을 반복

④ 각 부분 배열이 더 이상 분할되지 않으면 정렬이 완료

퀵 정렬(Quick Sort)

- 퀵 정렬 동작 방식



퀵 정렬(Quick Sort)

- 퀵 정렬 알고리즘

quickSort($A[], p, r$): ▷ $A[p...r]$을 정렬한다.

1. if ($p < r$)
    2. $q \leftarrow$ partition(A, p, r) ▷ 분할
    3. quickSort($A, p, q-1$) ▷ 왼쪽 부분 배열 정렬
    4. quickSort($A, q+1, r$) ▷ 오른쪽 부분 배열 정렬

partition($A[], p, r$):

배열 $A[p...r]$의 원소들을 기준 원소인 $A[r]$을 기준으로 양쪽으로 재배치하고 기준 원소가 자리한 위치를 리턴한다.

퀵 정렬

- 퀵 정렬 알고리즘

```
def quick_sort(arr):
    # 기본 조건: 배열이 길이 1 이하이면 이미 정렬된 상태
    if len(arr) <= 1:
        return arr

    pivot = arr[0]  # 첫 번째 요소를 피벗으로 선택
    left = [x for x in arr[1:] if x < pivot]   # 피벗보다 작은 값들
    right = [x for x in arr[1:] if x >= pivot] # 피벗보다 크거나 같은

    # 재귀적으로 정렬한 결과를 결합
    return quick_sort(left) + [pivot] + quick_sort(right)
```

실습 : 퀵 정렬(Quick Sort) 알고리즘 구현하기

- 퀵 정렬 알고리즘 파이썬 코드로 구현하기

```python
def quick_sort(arr):
    # 기본 조건: 배열이 길이 1 이하이면 이미 정렬된 상태
    if len(arr) <= 1:
        return arr

    pivot = arr[0]  # 첫 번째 요소를 피벗으로 선택(중앙값(arr[len(arr) // 2])
    left = [x for x in arr[1:] if x < pivot]   # 피벗보다 작은 값들
    right = [x for x in arr[1:] if x >= pivot] # 피벗보다 크거나 같은 값들

    # 재귀적으로 정렬한 결과를 결합
    return quick_sort(left) + [pivot] + quick_sort(right)

# 예시 사용
arr = [24, 9, 29, 14, 19, 27]
print("정렬된 배열:", quick_sort(arr))
```

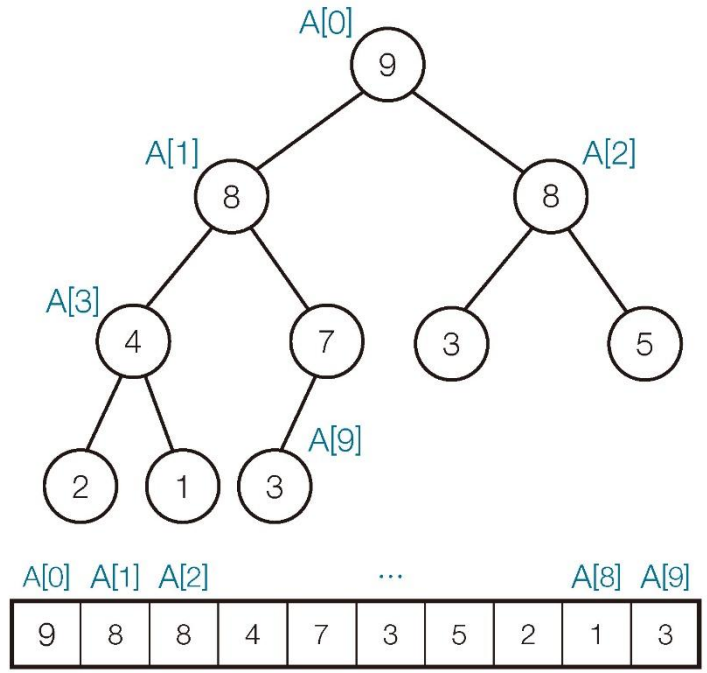
힙 정렬(Heap Sort)

주어진 배열을 힙(heap)으로 만든 다음,
차례로 하나씩 힙(heap)에서 제거함으로써 정렬을 수행하는 알고리즘

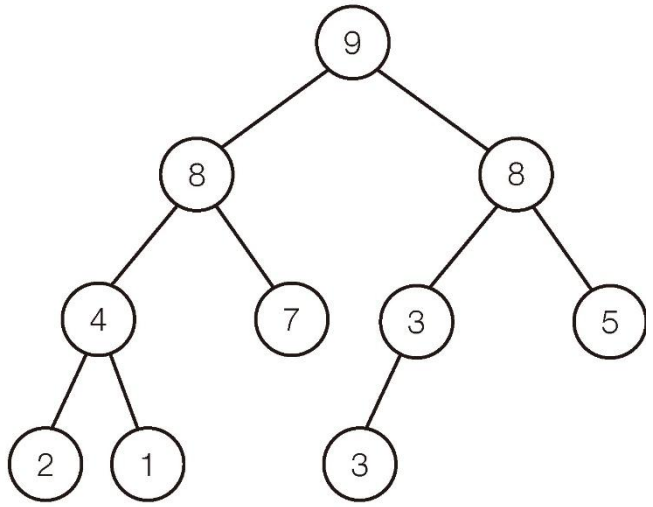
- 힙(Heap) 자료구조를 기반으로 힙의 특성을 이용한 정렬 알고리즘
- 힙은 완전 이진 트리(Complete Binary Tree)로 구성되어 있으며, 노드의 값이 자식 노드의 값보다 항상 크거나(Max-heap) 작은(Min-heap) 속성을 가지는 이진 트리
- 비교 기반 제자리 정렬(in-place sorting)의 특성
- 힙 구성 + 힙 정렬
- 최선/최악 시간 복잡도: $O(n \log n)$

힙 (Heap)

- 완전 이진 트리(Complete Binary Tree)를 사용한다.
- 힙 특성(Property): 모든 노드는 값을 갖고, 자식 노드(들) 값보다 크거나 같다



(a) 10개의 원소로 구성된 힙과 대응되는 배열



(b) 힙 조건 ❷는 만족하지만 힙 조건 ❶을 만족하지 않아 힙이 아닌 예

힙 정렬

- ## 힙 정렬 동작 방식

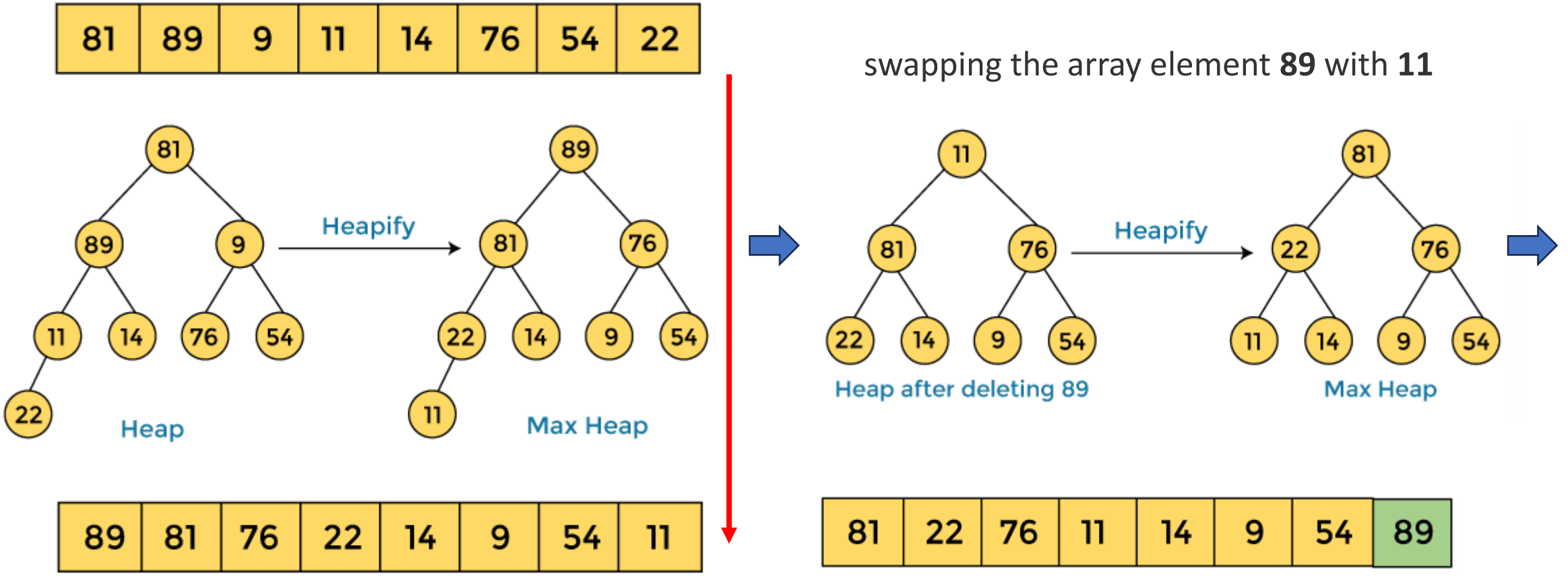
① 주어진 배열을 최대 힙(Max Heap) 또는 최소 힙(Min Heap)으로 만든다. (heapify)
(이 과정에서는 주어진 배열을 이진 힙 트리로 변환)

② 최대 힙 또는 최소 힙에서 최상단의 루트 노드를 제거하고, 이를 배열의 가장 뒤쪽으로 이동시킴. (이로써 배열의 가장 큰 값 또는 가장 작은 값이 제거되고, 정렬된 부분 배열에 추가됨)

③ 힙의 크기를 줄이고, 남은 요소들에 대해 다시 힙을 구성하여 정렬을 진행
(이 과정을 반복하여 정렬이 완료될 때까지 수행)

힙 정렬

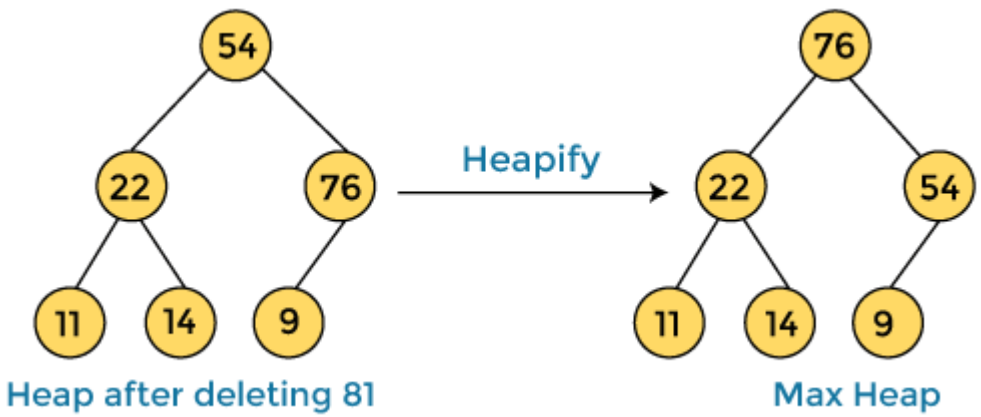
- 힙 정렬 동작방식



힙 정렬

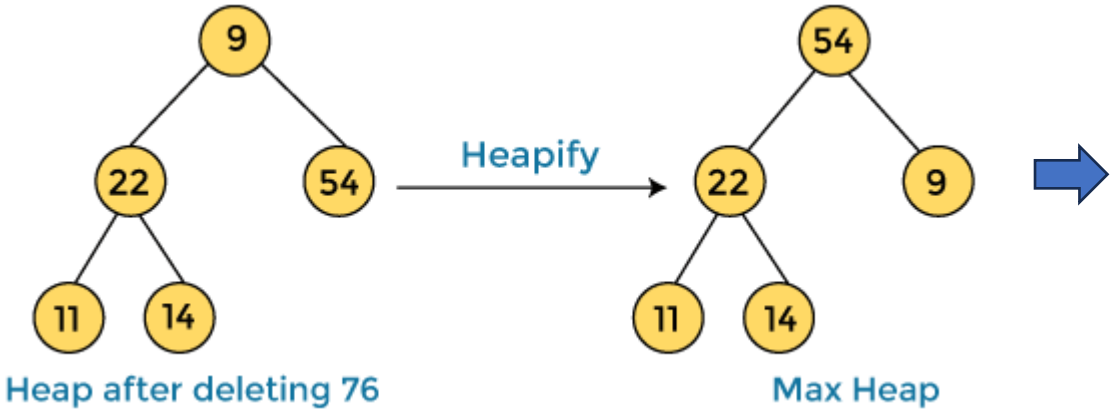
▪ 힙 정렬 동작방식

swapping the array element **81** with **54**



76	22	54	11	14	9	81	89

swapping the array element **76** with **9**

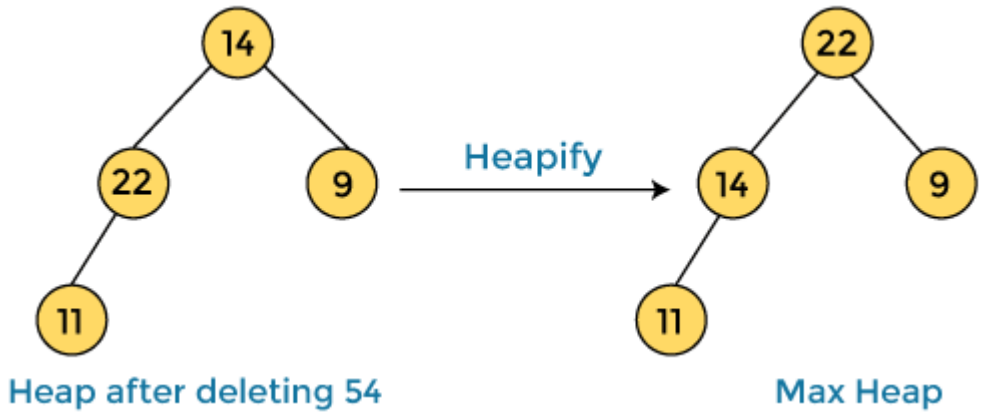


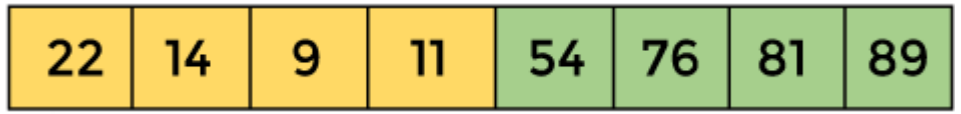
54	22	9	11	14	76	81	89

힙 정렬

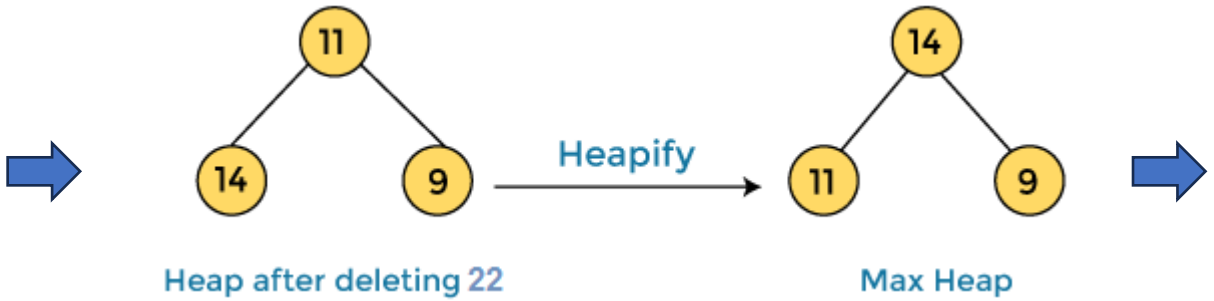
▪ 힙 정렬 동작방식

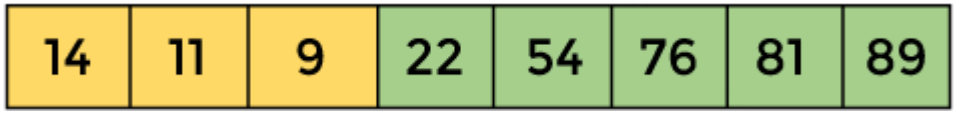
swapping the array element **54** with **14**





swapping the array element **22** with **11**

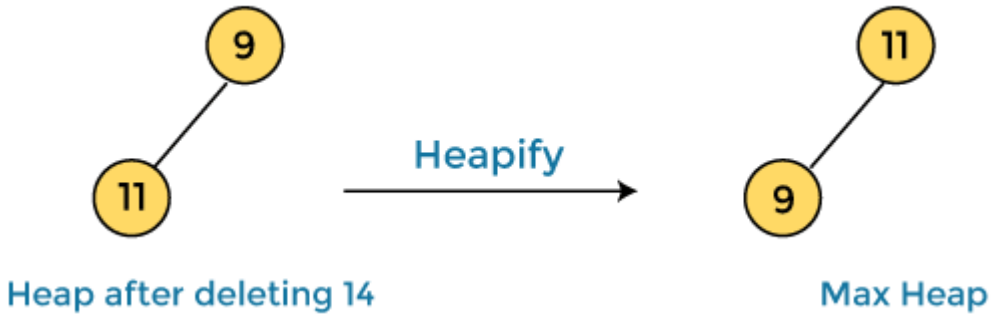


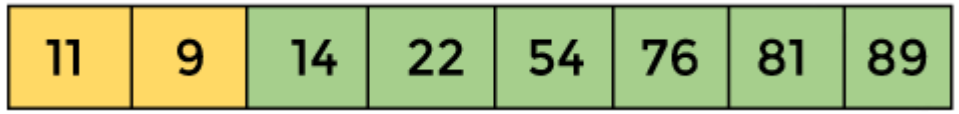


힙 정렬

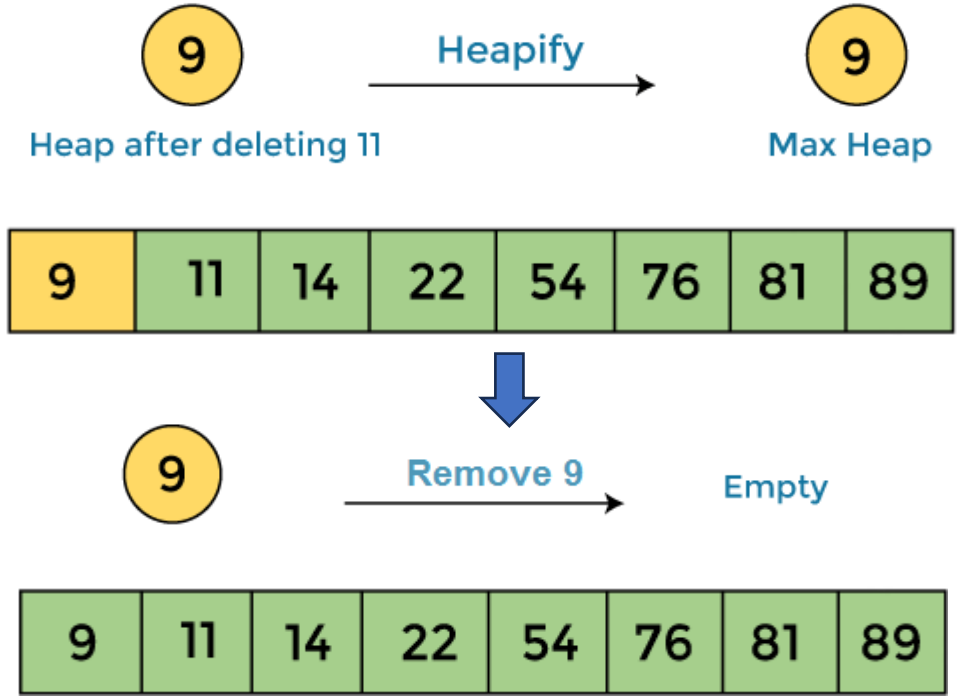
- 힙 정렬 동작 방식

swapping the array element **14** with **9**





swapping the array element **11** with **9**



힙 정렬

- 힙 정렬 알고리즘

```
def heap_sort(arr):
    n = len(arr)

    # 최대 힙 생성
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # 최대 힙에서 요소를 하나씩 추출하여 정렬
    for i in range(n - 1, 0, -1):
        # 루트 노드(최대값)와 마지막 노드를 교환
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)  # 변경된 부분 힙에 대해 heapify 호출
```

```python
def heapify(arr, n, i):
    largest = i  # 루트 노드 설정
    left = 2 * i + 1  # 왼쪽 자식 노드
    right = 2 * i + 2  # 오른쪽 자식 노드

    # 왼쪽 자식 노드가 존재하고, 더 큰 값을 가지면
    if left < n and arr[left] > arr[largest]:
        largest = left

    # 오른쪽 자식 노드가 존재하고, 더 큰 값을 가지면
    if right < n and arr[right] > arr[largest]:
        largest = right

    # 최대 힙 속성을 만족하지 않으면 루트 노드와 가장 큰 자식 노
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        # 변경된 부분 힙에 대해 재귀적으로 heapify 호출
        heapify(arr, n, largest)
```

셸 정렬(Shell Sort)

- 셸 정렬(Shell Sort)
  - 삽입 정렬(Insertion Sort)을 개선한 정렬 알고리즘 중 하나

    (삽입 정렬은 배열의 거의 정렬된 상태에서 효율적으로 동작하지만, 배열이 큰 경우에는 비효율적)
  - 삽입 정렬의 단점을 개선하기 위해 제안된 알고리즘으로, 배열을 일정한 간격으로 나누어 삽입 정렬을 수행하는 방식

    Hibbard, Shell, Knuth 등의 간격 시퀀스
  - 셸 정렬은 삽입 정렬보다 훨씬 빠르며, 시간 복잡도는 간격 시퀀스에 따라 달라지지만 보통 O(n log n) 이하임, 제자리 정렬(in-place sorting)
  - 최선 시간 복잡도: $O(n \log n)$
  - 최악 시간 복잡도: $O(n^2)$

셸 정렬

- ## 셸 정렬 동작 방식

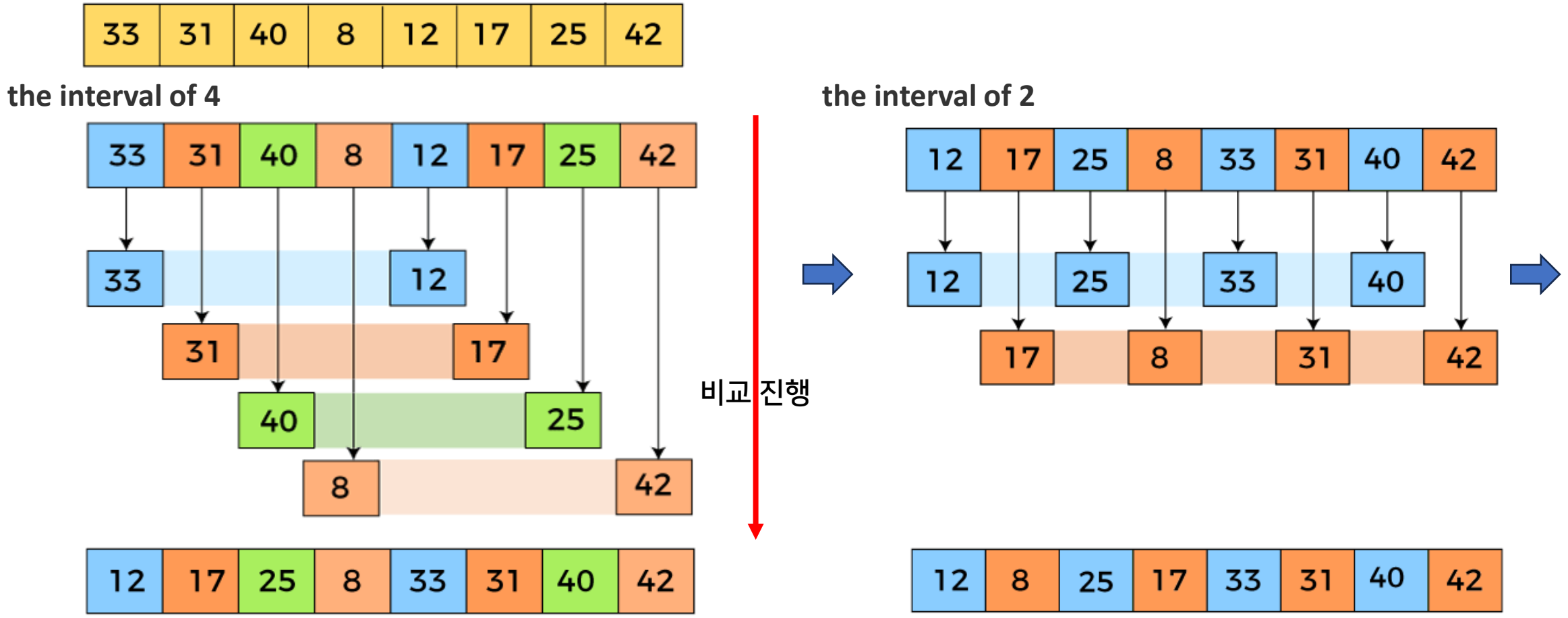
① 일정한 간격(간격 시퀀스)을 선택하여 배열을 부분적으로 정렬

② 선택된 간격에 따라 배열을 여러 부분 배열로 분할하고, 각 부분 배열에 대해 삽입 정렬을 수행.

③ 간격을 줄여가면서 위의 과정을 반복하여 정렬을 완료

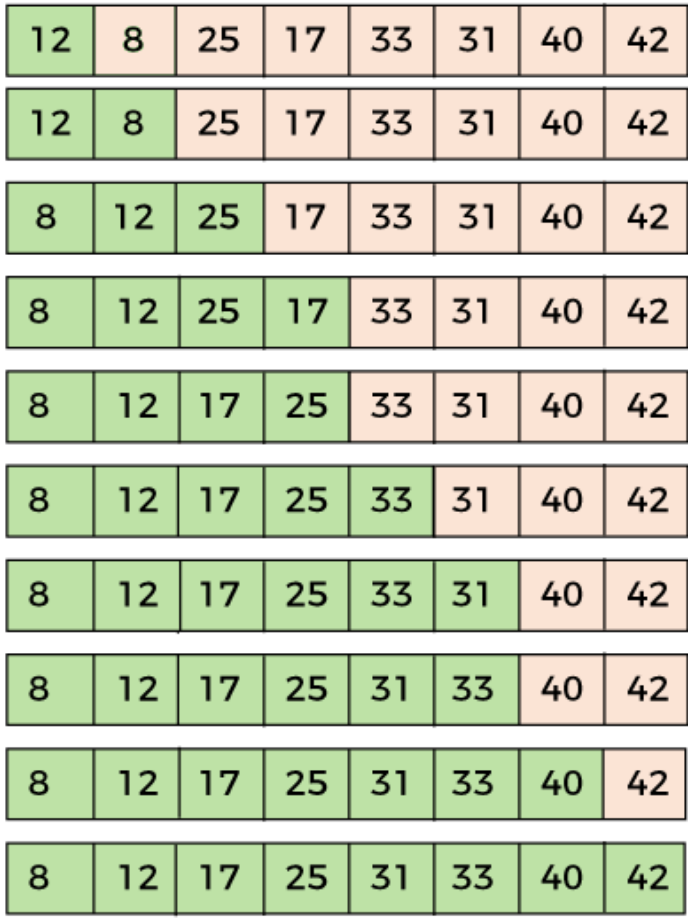
셸 정렬

- 셸 정렬 동작 방식



셸 정렬

- 셸 정렬 동작방식



삽입 정렬(Insertion Sort) 진행

셸 정렬

- 셸 정렬 알고리즘

```
def shell_sort(arr):
    n = len(arr)
    gap = n // 2  # 초기 간격 설정

    # 간격을 줄여가면서 반복
    while gap > 0:
        # 삽입 정렬을 사용하여 각 부분 배열을 정렬
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
            print(f"gap: {gap} arr: {arr}")
        gap //= 2  # 간격을 줄임
```

특수 정렬 알고리즘

특수 정렬 알고리즘

- 특수 정렬 알고리즘
  - 원소들이 **특수한 성질을 만족(특정 조건에서)**하면 Θ(n) 정렬도 가능하다
    - 계수 정렬(Counting Sort)
    - 기수 정렬(Radix Sort)
    - 버킷 정렬(Bucket Sort)
  - 이 알고리즘들은 비교 기반의 정렬 방법이 아니며, 특정 조건하에서 선형 시간에 가까운 성능을 보임
    - ex: 계수 정렬과 기수 정렬은 정수나 작은 범위의 숫자를 정렬할 때 매우 효율적임

계수 정렬

- 계수 정렬(Counting Sort)
  - **정수 배열을 정렬**하는 정렬 알고리즘. 비교 기반 정렬 알고리즘이 아님
  - 정수 배열에 등장하는 각 원소의 개수를 세고, 이를 이용하여 정렬하는 방식으로 동작
  - 계수 정렬은 원소들의 범위가 제한되어 있을 때 특히 효율적으로 동작
  - 원소들의 크기가 모두 – O(n) ~ O(n+k) 범위에 있을 때
  - 최선/최악 시간 복잡도: O(n + k), k는 데이터의 범위
    - 최대 원소를 찾는다.

계수 정렬

- ## 계수 정렬 동작 방식

① 정수 배열에 등장하는 각 원소의 개수를 세고, 해당 원소의 개수를 저장하는 카운트 배열을 생성

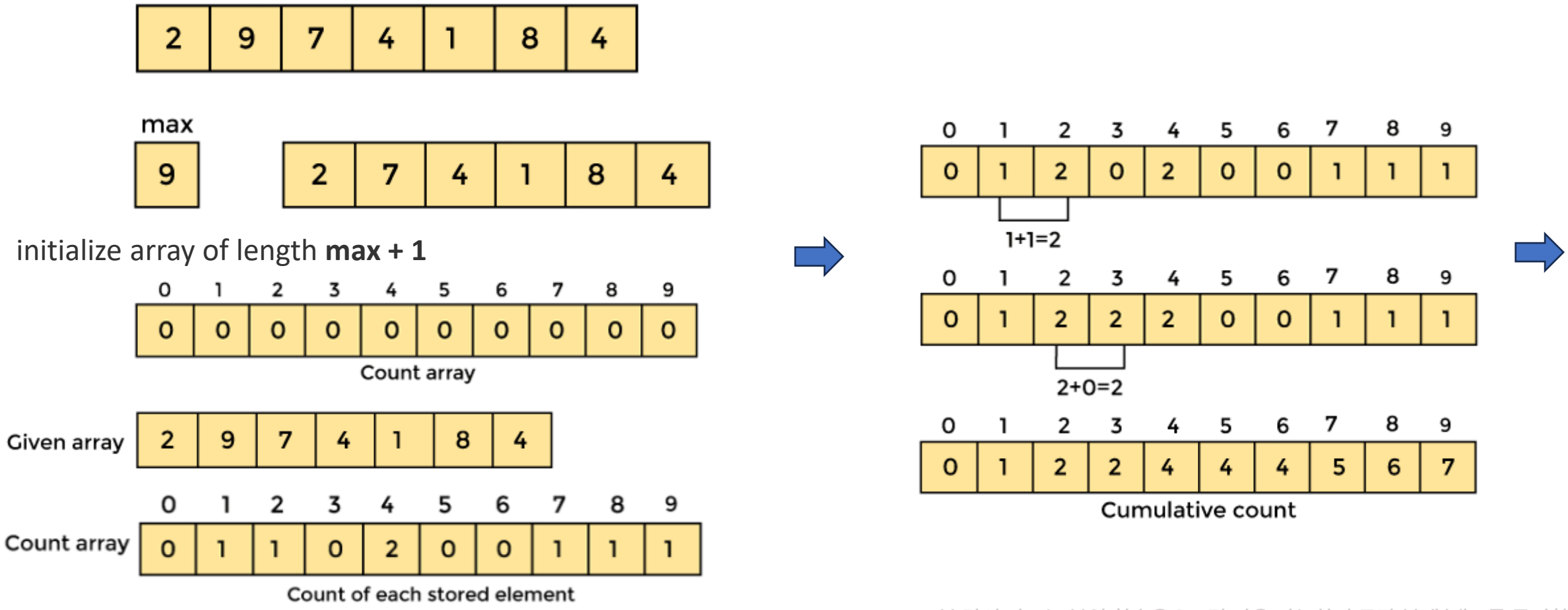
② 카운트 배열을 사용하여 정렬된 배열을 생성

(이때, 원소의 값과 해당 원소의 개수를 이용하여 정렬)

③ 정렬된 배열을 반환

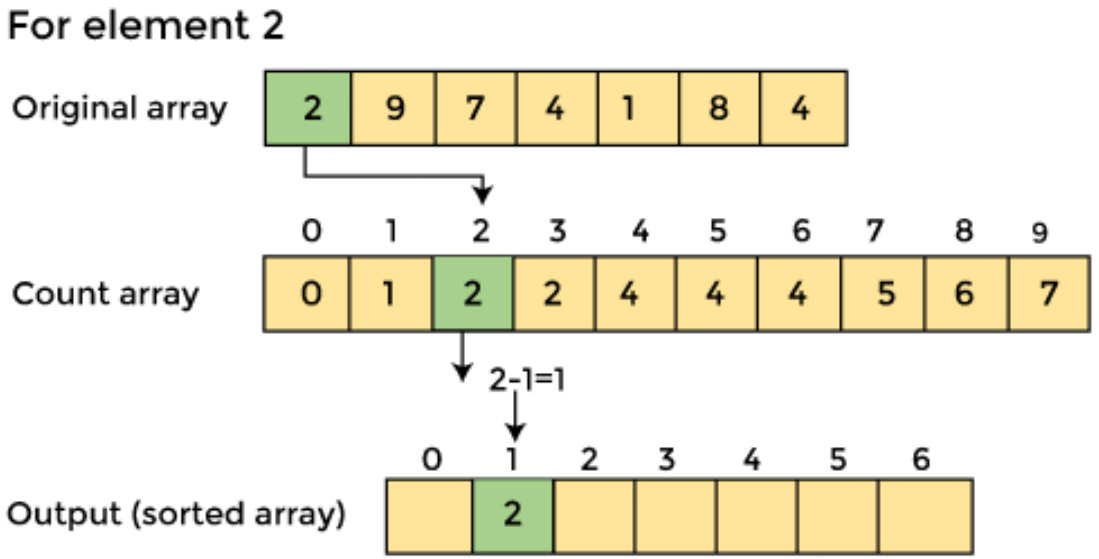
계수 정렬

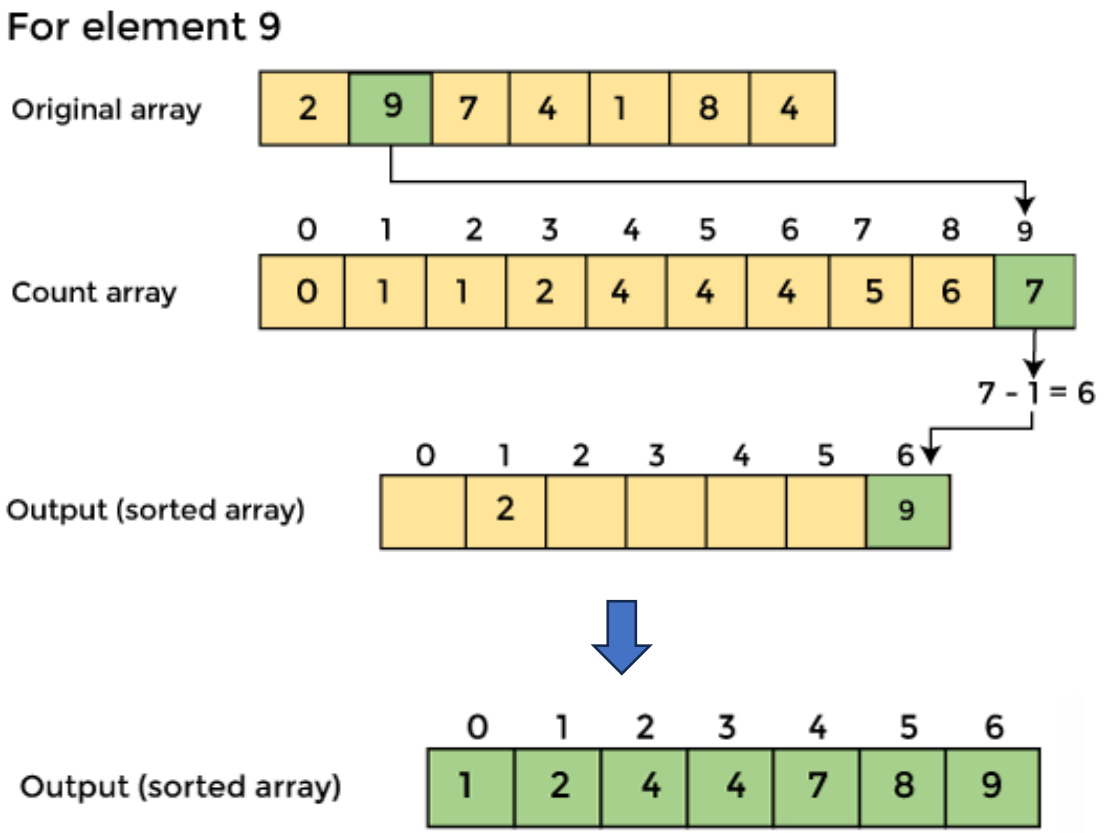
- 계수 정렬 동작 방식



계수 정렬

- 계수 정렬 동작 방식





계수 정렬

- 계수 정렬 알고리즘

```
def counting_sort(arr):
    # 배열의 최댓값을 찾아 카운트 배열을 생성합니다.
    max_val = max(arr)
    count = [0] * (max_val + 1)

    # 각 원소의 개수를 카운트합니다.
    for num in arr:
        count[num] += 1

    # 정렬된 배열을 저장할 리스트를 생성합니다.
    sorted_arr = []

    # 카운트 배열을 사용하여 정렬된 배열을 생성합니다.
    for i in range(len(count)):
        sorted_arr.extend([i] * count[i])

    return sorted_arr
```

기수 정렬

- 기수 정렬(Radix Sort)
  - 정수나 문자열과 같은 키(key)를 가진 요소들을 정렬하는 데 사용
  - 각 요소를 자릿수별로 비교하지 않고, 키의 각 자릿수를 사용하여 정렬하는 방식
  - 비교 기반 정렬 알고리즘이 아님
  - 기수 정렬은 각 자릿수를 기준으로 정렬하기 때문에 비교 기반 정렬 알고리즘보다 효율적으로 동작할 수 있음
  - 원소들이 모두 k 이하의 자릿수를 가졌을 때 (k: 상수)
  - 최선/최악 시간 복잡도: O(kn), k는 자릿수

기수 정렬

- ## 기수 정렬 동작 방식

① 가장 낮은 자릿수부터 가장 높은 자릿수까지 순차적으로 정렬

② 각 자릿수를 기준으로 요소들을 버킷(bucket)에 나누어 저장

* 각 자릿수를 기준으로 나누는 방식

- 누적 분배(least significant digit, LSD)

- 가장 유의미한 자릿수(most significant digit, MSD) → 반대 방향

③ 각 버킷에 저장된 요소들을 순차적으로 다시 수집하여 정렬된 배열을 생성

(이 과정을 가장 높은 자릿수까지 반복)

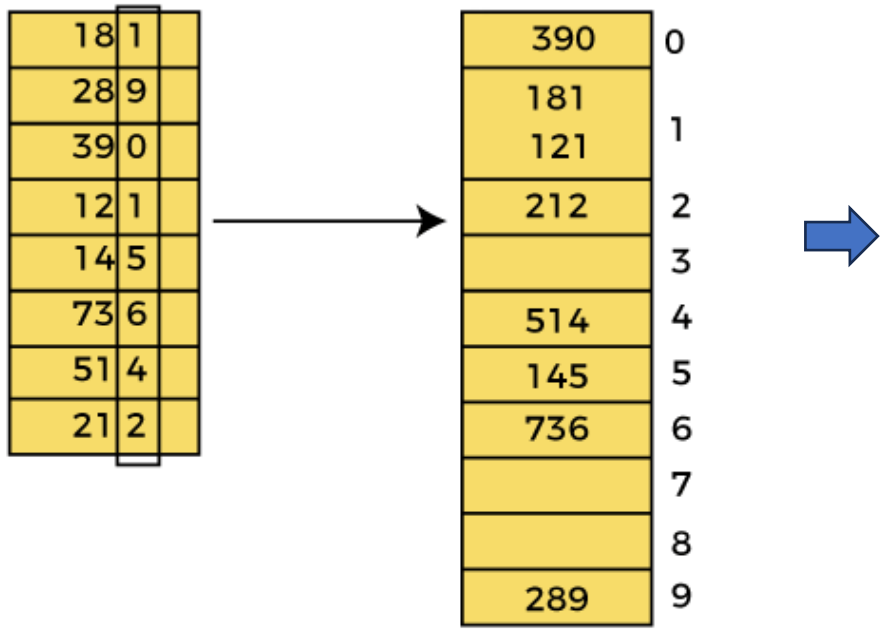
기수 정렬

- 기수 정렬 동작 방식

181	289	390	121	145	736	514	212

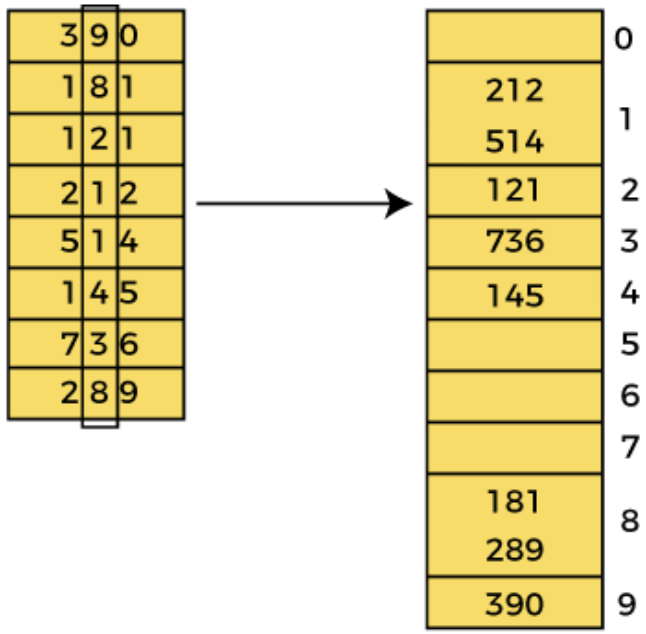
the largest element is **736** that have **3** digits in it.
So, the loop will run up to three times

Pass 1:



390	181	121	212	514	145	736	289

Pass 2:



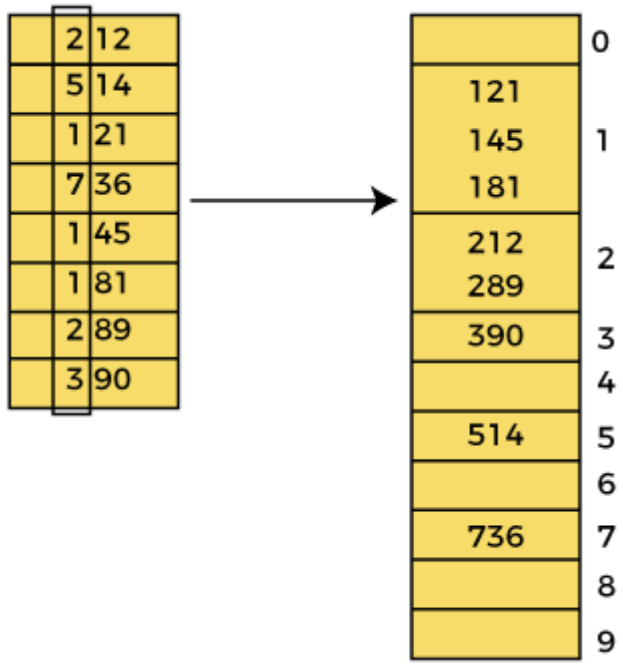
212	514	121	736	145	181	289	390

기수 정렬

- 기수 정렬 동작 방식

212	514	121	736	145	181	289	390

Pass 3:



121	145	181	212	289	390	514	736

기수 정렬

- 기수 정렬 알고리즘

```python
def radix_sort(arr):
    # 최대값 찾기
    max_val = max(arr)

    # 최대값을 기준으로 각 자릿수에 대해 counting sort 수행
    exp = 1
    while max_val // exp > 0:
        counting_sort(arr, exp)
        exp *= 10
```

```python
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    # 각 자릿수에 해당하는 값의 등장 횟수를 세기
    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    # 등장 횟수를 누적 합으로 변경
    for i in range(1, 10):
        count[i] += count[i - 1]

    # output 배열에 요소를 정렬하여 배치
    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    # 원래 배열로 복사
    for i in range(n):
        arr[i] = output[i]
```

버킷 정렬

- 버킷 정렬(Bucket Sort)
  - 원소들이 균등 분포(Uniform Distribution):배열이 일정한 범위 내에 분포되어 있는 경우를 이룰 때 유용한 정렬 알고리즘
  - 입력 배열을 여러 개의 버킷으로 나눈 다음, 각 버킷을 개별적으로 정렬하여 최종적으로 정렬된 결과를 얻는 알고리즘
  - 입력 배열의 분포가 불균형할 경우에는 성능이 저하될 수 있
  - 최선 시간 복잡도: O(n + k), k는 버킷의 개수
  - 최악 시간 복잡도: $O(n^2)$

버킷 정렬

- ## 버킷 정렬 동작 방식

① 입력 배열을 고정된 개수의 버킷으로 나눈다.

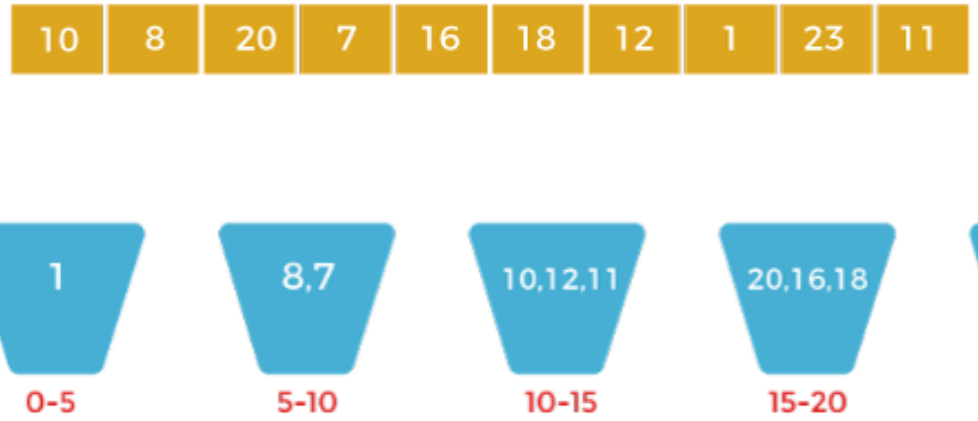
(이때, 각 버킷의 범위는 입력 배열의 값의 범위에 따라 결정)

② 각 버킷에 대해 정렬 알고리즘(보통은 삽입 정렬이나 병합 정렬 등)을 사용하여 정렬

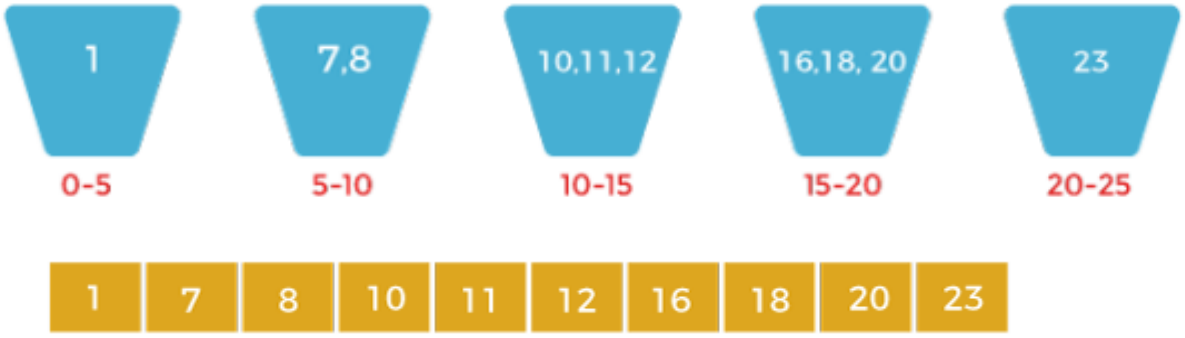
③ 각 버킷에서 정렬된 요소들을 다시 원래 배열에 합친다.

버킷 정렬

- 버킷 정렬 동작 방식



sort each bucket individually



버킷 정렬

- 버킷 정렬 알고리즘

```python
def insertion_sort(bucket):
    for i in range(1, len(bucket)):
        key = bucket[i]
        j = i - 1
        while j >= 0 and bucket[j] > key:
            bucket[j + 1] = bucket[j]
            j -= 1
        bucket[j + 1] = key
```

```python
def bucket_sort(arr):
    # 입력 배열의 최댓값 찾기
    max_val = max(arr)
    # 버킷 개수 결정
    num_buckets = len(arr)
    # 각 버킷 초기화
    buckets = [[] for _ in range(num_buckets)]
    # 각 요소를 적절한 버킷에 할당
    for num in arr:
        index = num * num_buckets // (max_val + 1)
        buckets[index].append(num)
    # 각 버킷 정렬
    for bucket in buckets:
        insertion_sort(bucket)
    # 정렬된 버킷을 합쳐 최종 결과 생성
    k = 0
    for bucket in buckets:
        for num in bucket:
            arr[k] = num
            k += 1
```

실습문제 : 특수 정렬 알고리즘 구현하기

- 앞에서 확인한 특수 정렬 알고리즘을 파이썬으로 구현하세요.
  1) Counting Sort
  2) Radix Sort
  3) Bucket Sort

Q & A

Next Topic

- 정렬 알고리즘2(다양한 데이터 사용한 정렬 실습)

Keep learning, see you soon!