ChatGPT

알고리즘 스터디: 5주차 정렬 알고리즘

새로운 주제로 들어가기 전에, **4주차에 배운 내용을 간략히 되짚어** 보겠습니다. 지난 시간에는 **완전탐색(brute-force)**, **재귀(recursion)**, **분할 정복(divide and conquer)**, **백트래킹(backtracking)** 등의 핵심 알고리즘 설계 기법을 다루었습니다. 완전탐색은 가능한 모든 경우를 시도하여 해답을 찾는 기법으로, 입력이 작을 때는 확실한 결과를 보장하지만 입력 크기가 커지면 지수적 시간 증가로 비현실적이 될 수 있다는 점을 확인했습니다. 재귀는 문제를 자기 자신을 호출하여 작은 하위 문제로 분할해 푸는 접근으로, 기저 조건과 재귀적 단계의 개념을 배웠습니다. 분할 정복은 문제를 독립적인 부분 문제로 나눈 뒤 결과를 합쳐 효율적으로 해결하는 방법으로, 예시로 배열 정렬(병합 정렬)이나 거듭제곱 계산 등의 시간 복잡도를 크게 향상시킬 수 있음을 살펴보았습니다. 백트래킹은 해답 후보를 단계적으로 구축하면서 조건에 맞지 않는 경로는 조기에 가지치기하여 탐색하는 기법으로, 퍼즐이나 조합 문제(예: N-퀸 문제)에서 유용하게 활용되는 것을 배웠습니다. 이러한 다양한 알고리즘 전략을 숙지함으로써, 주어진 문제에 적합한 접근법을 선택하고 더 효율적인 해법을 설계하는 능력을 키울 수 있었습니다.

정렬 알고리즘 소개

이번 주에는 **정렬(Sorting)** 알고리즘을 집중적으로 학습합니다. **정렬**은 컴퓨터과학에서 가장 기본적이면서도 중요한 문제 중 하나입니다. 주어진 데이터 **집합을 일정한 순서대로 배열**하는 작업은, 그 자체로도 유용할 뿐 아니라 다른 알고리즘의 전처리 단계로 자주 활용됩니다. 실제로 정렬 문제는 간단한 문제 설명에도 불구하고 효율적으로 풀기 위해 **오랫동안 많은 연구**가 이루어져 왔습니다 [1]. 비교 기반의 정렬 알고리즘은 이론적으로 최소 $\Omega(n \log n)$의 비교 연산이 필요하며 [2], 이러한 하한에 근접하면서도 빠르게 동작하는 알고리즘들이 개발되어 왔습니다. 정렬 알고리즘은 **빅오 표기**, **분할 정복**, **힙 트리** 등의 핵심 개념을 배우기 위한 좋은 예제로도 잘 알려져 있어 대부분의 입문 **컴퓨터과학 수업에서 중요하게 다뤄집니다** [3]. 따라서 정렬 알고리즘을 학습하는 것은 **자료구조**, **알고리즘 분석**, **문제 해결** 측면에서 큰 의미가 있습니다.

정렬 알고리즘을 논할 때는 몇 가지 **중요한 개념과 속성**을 짚고 넘어가야 합니다:

- **안정성 (Stability):** 정렬 알고리즘이 **안정적**이라는 것은, **동일한 키 값을 가진 요소들의 상대적 순서가 정렬 후에도 유지됨**을 의미합니다 [4]. 안정 정렬을 사용하면 여러 키로 이루어진 데이터를 1차 키로 정렬한 뒤 2차 키로 정렬할 때 유용합니다. 예를 들어 학생 명부를 이름으로 정렬한 후 반별로 다시 정렬할 때 안정 정렬을 사용하면, 같은 반에서는 기존의 이름 순서가 유지되어 사람이 읽기 쉬운 결과를 얻을 수 있습니다. 반면 **불안정(unstable)** 정렬은 동일 값의 순서를 보장하지 않으며, 필요하다면 원본 인덱스를 비교에 포함시키는 방식으로 **안정성을 부여**할 수 있지만 추가 비용이 듭니다.

- **제자리 여부 (In-place): 제자리 정렬**은 정렬을 위해 **배열 외에 추가적인 메모리를 거의 사용하지 않는** 알고리즘을 말합니다. 보통 $O(1)$의 추가 공간만 사용하면 제자리 정렬로 간주되며 [5], 일부 문헌에서는 재귀 호출 스택으로 인한 $O(\log n)$ 공간도 허용 범위에 넣기도 합니다. 제자리 정렬은 메모리 사용을 최소화하기 때문에 메모리 제약이 있는 환경에서 유리합니다. 다만, 제자리 정렬 여부는 알고리즘 구현에 따라 달라질 수 있으며 (예를 들어 병합 정렬은 배열을 사용할 때 기본적으로 제자리 정렬이 아니지만, 연결 리스트에 적용하면 추가 메모리가 거의 필요 없습니다), 일반적으로 **삽입 정렬, 선택 정렬, 퀵 정렬, 힙 정렬** 등은 제자리 알고리즘이고, **병합 정렬, 계수 정렬, 기수 정렬** 등은 추가 버퍼가 필요한 알고리즘입니다.

- **비교식 vs. 비비교식: 비교 기반(comparison-based)** 정렬은 원소들을 비교 연산(<, > 등)을 통해 정렬을 수행합니다. 이러한 알고리즘에는 **$O(n \log n)$** 시간 복잡도의 하한이 존재하며, 퀵 정렬, 병합 정렬, 힙 정렬 등 대부분의 표준 정렬이 여기에 속합니다 [2]. 반면 **비비교식(non-comparison)** 정렬은 직접적인 비교 연산 대신 다른 방법으로 정렬을 수행하여 이론상의 시간 하한을 피할 수 있습니다. 대표적으로 **계수 정렬**이나 **기수 정렬**은 키 값을 인덱스로 사용하거나 여러 자릿수를 분리 처리함으로써 비교를 생략하고, **선형 시간대의 정렬도 가**

능합니다. 다만 이러한 알고리즘들은 적용 가능한 데이터의 형태나 조건(예: 정수 범위, 자릿수 길이 등)에 제한이 있고, 추가 메모리를 사용한다는 trade-off가 있습니다.

- **정렬 대상의 구조:** 정렬은 배열과 같은 **선형 자료구조**에서 가장 흔히 논의되지만, 그 외의 구조나 큰 데이터에 대한 정렬도 중요합니다. 예를 들어 **연결 리스트**를 정렬할 때는 임의 접근(random access)이 불가능하므로 일반적인 퀵 정렬이나 힙 정렬보다 **병합 정렬**이 적합합니다 (연결 리스트에서는 병합 정렬이 **제자리 구현**도 가능하기 때문입니다 [6]). 또한 메모리에 한 번에 다 담을 수 없는 **외부 정렬**의 경우, 테이프나 디스크에 있는 데이터를 **병합 정렬** 방식으로 부분 정렬 후 합치는 기법이 쓰입니다. 이처럼 **정렬 대상 데이터의 특성**(메모리 크기, 데이터 접근 방식 등)에 따라 적절한 정렬 알고리즘을 선택해야 합니다.

요약하면, **정렬 알고리즘**은 다양한 조건과 요구사항에 맞게 여러 종류가 존재하며, 각 알고리즘은 **시간 복잡도**, **공간 복잡도**, **안정성** 측면에서 장단점을 가집니다. 이제부터 자주 쓰이고 중요한 정렬 알고리즘들을 시간 복잡도별로 나누어 자세히 살펴보겠습니다.

O(n^2) 시간 복잡도의 정렬 알고리즘

먼저 비교적 구현이 쉽지만 효율은 떨어지는 O(n^2) 시간 복잡도의 정렬 알고리즘들을 알아보겠습니다. 여기에 해당하는 **선택 정렬**, **버블 정렬**, **삽입 정렬**은 모두 이중 루프를 사용하기 때문에 입력 크기가 커지면 성능이 급격히 저하되지만, **알고리즘 교육용**으로 자주 다뤄지고 작은 입력에서는 사용할 수 있습니다. 이들은 서로 유사한 O(n^2)이라도 미묘한 차이가 있으므로 각각의 특징을 정리합니다.

선택 정렬 (Selection Sort)

선택 정렬은 가장 간단한 정렬 알고리즘 중 하나로, 이름 그대로 매 단계마다 **가장 작은 원소를 선택하여** 앞쪽으로 보내는 방식을 취합니다. 구체적으로, 배열에서 **최솟값을 찾아 첫 번째 위치와 교환**하고, 다음에는 남은 부분에서 최솟값을 찾아 두 번째 위치와 교환하는 식으로 진행됩니다. 이렇게 하면 반복 횟수마다 **확정된 최솟값들이 맨 앞부터 순서대로 쌓이게** 됩니다.

- **동작 과정:** 선택 정렬은 내부 루프로 최솟값 탐색, 외부 루프로 선택 및 교환을 수행합니다. 구현은 비교적 쉬우며, 배열 내에서 교환(swap)을 통해 제자리로 동작합니다. 다만, **정렬이 이미 어느 정도 되어 있어도 모든 쌍을 비교**하므로 **데이터 분포에 상관없이 항상 동일한 작업**을 합니다 (어떤 경우에도 개선되지 않음).

- **파이썬 구현:** 선택 정렬은 제자리 정렬이므로 별도 반환값 없이 리스트 자체를 정렬하도록 구현할 수 있습니다. 예를 들어:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # i 위치에 현 단계 최솟값을 놓는다
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# 사용 예시
data = [64, 25, 12, 22, 11]
selection_sort(data)
print(data)   # [11, 12, 22, 25, 64]
```

위 코드에서 이중 루프를 통해 매번 최소값의 인덱스 `min_idx`를 찾은 뒤 한 번의 교환을 수행합니다. 총 n번의 교 환이 일어나며, 비교 연산은 n(n-1) / 2번 수준으로 발생합니다.

- **복잡도:** 선택 정렬의 **시간 복잡도**는 최선, 평균, 최악의 경우 모두 **O(n^2)**입니다. 데이터가 이미 정렬되어 있더라도 내부 루프에서 모든 나머지 원소와의 비교를 수행하므로 비용이 줄어들지 않습니다 [7] [8]. 한편 **공간 복잡도**는 O(1)로, 추가 배열이 필요 없는 **제자리 알고리즘**입니다 [9].

- **안정성:** 선택 정렬은 기본 형태로 구현하면 **안정적이지 않은** 정렬입니다. 예를 들어 `[3, 3, 1]` (화살표로 동일값 표기) 같은 입력에서, 첫 번째 패스에 최소값 `1`을 찾은 뒤 위치 0과 교환하면 두 `3`의 상대순서가 바뀌게 됩니다. 이러한 이유로 **표준 선택 정렬은 불안정**하며, 이를 안정적으로 만들려면 교환 대신 **삽입**이나 **링크드 리스트**를 활용하는 등의 추가 조치가 필요합니다 [10]. 그러나 대부분 실용적인 상황에서는 굳이 선택 정렬의 안정 버전을 사용하지는 않습니다.

- **특징 및 용도:** 선택 정렬은 **구현이 매우 간단**하다는 장점이 있습니다. 또한 각 단계마다 정확히 한 번의 교환만 일어나므로, **교환 횟수가 최소**입니다. 이러한 특성 때문에 **쓰기(write) 연산 비용이 큰 환경(예: 플래시 메모리)**에서 쓰기 횟수를 줄이기 위해 간혹 사용되기도 합니다. 그러나 비교 연산 자체는 많이 일어나기 때문에, 전체적인 속도 면에서는 삽입 정렬 등 다른 O(n^2) 알고리즘보다도 대체로 느린 편입니다 [8]. 요약하면 선택 정렬 은 **메모리 제약이 있거나 구현 난이도를 최우선**으로 할 때 고려할 수 있지만, **속도가 중요할 때는 잘 쓰이지 않 는** 알고리즘입니다.

버블 정렬 (Bubble Sort)

버블 정렬은 인접한 원소들을 비교하여 크기가 순서에 맞지 않으면 서로 교환(swap)하는 과정을 반복함으로써 정렬을 이루는 알고리즘입니다. 이 과정에서 큰 값은 거품이 수면으로 올라오듯이 배열의 뒷부분으로 이동하고, 작은 값은 앞쪽으로 이동하게 됩니다. 버블 정렬은 항상 배열의 앞에서부터 인접한 쌍을 검사해가며 필요시 교환하므로, 한 **패스(pass)**가 끝나면 가장 큰 값이 배열 끝에 정렬됩니다.

- **동작 과정:** 배열을 **여러 번 반복 통과**(pass)합니다. 각 패스에서는 인접한 두 원소를 차례로 비교하여, **앞 원소가 뒤 원소보다 크면 자리 교환**을 합니다. 1회 패스가 끝나면 가장 큰 원소가 맨 뒤로 이동하므로, 다음 패스에서는 마지막 원소는 검사하지 않아도 됩니다. 이런 식으로 패스를 반복하며, 더 이상 교환이 필요 없을 때까지 진행합니다.

- **파이썬 구현:** 버블 정렬도 제자리 정렬로 구현됩니다. 교환이 일어나지 않으면 정렬이 완료된 것이므로, 이를 체크하여 루프를 조기 종료할 수도 있습니다:

```
def bubble_sort(arr):
    n = len(arr)
    for end in range(n - 1, 0, -1):
        swapped = False
        for i in range(end):
            if arr[i] > arr[i + 1]:
                arr[i], arr[i + 1] = arr[i + 1], arr[i]
                swapped = True
        if not swapped:
            break

# 사용 예시
data = [5, 1, 4, 2, 8]
```

```
bubble_sort(data)
print(data)   # [1, 2, 4, 5, 8]
```

위 코드에서 `end`는 아직 정렬되지 않은 구간의 끝 인덱스를 의미하며, 내부 루프는 `0`부터 `end`까지 인접 요소를 비교합니다. 한 번의 패스 후에는 가장 큰 값이 `end` 위치로 "버블링"되므로 `end`를 하나씩 줄여가며 반복합니다. `swapped` 플래그를 통해 **교환이 한 번도 발생하지 않으면 루프를 조기 종료**하여 최선의 경우 시간 복잡도를 개선합니다.

- **복잡도:** 버블 정렬의 **평균 및 최악 시간 복잡도**는 다른 단순 정렬들과 마찬가지로 **O(n^2)**입니다 [11]. 이중 루프로 인해 n이 커지면 실행 시간이 급격히 증가합니다. 그러나 주목할 점은 **버블 정렬은 입력이 거의 정렬 되어 있는 경우 매우 효율적**이라는 것입니다. 위 구현처럼 교환 발생 여부를 체크하면, **이미 정렬된 리스트에 대 해서는 한 번의 패스만에 종료되어 O(n)**만에 처리가 가능합니다 [12]. 이는 **버블 정렬이 가지는 적응성 (adaptivity)**으로, 입력이 정렬되어 있거나 부분적으로 정렬되어 있을 때 시간 복잡도가 평균보다 크게 향상됩니다. 공간 복잡도는 상수 공간만을 사용하므로 **제자리 알고리즘**입니다.

- **안정성:** 버블 정렬은 인접 요소의 비교 및 교환만 수행하므로 **안정적인 정렬 알고리즘**입니다 [13]. 값이 동일한 두 원소가 있다면 교환 조건 `>`에서 등호를 포함하지 않음으로써(즉, 앞의 것이 뒤보다 클 때에만 교환) 그들의 상대적인 순서는 유지됩니다. 위 구현에서도 `if arr[i] > arr[i+1]`으로 되어 있어, 동일한 값일 경우 교환이 일어나지 않습니다. 따라서 입력에서 같은 값을 가진 원소들의 순서가 결과에서도 그대로 유지됩니다.

- **특징 및 용도:** 버블 정렬은 알고리즘이 **직관적**이고 구현이 쉬워 교육용으로 자주 소개됩니다. 특히 한 패스마다 가장 큰 값이 뒤로 확정되는 동작은 시각적으로도 이해하기 쉽습니다. 그러나 성능 면에서는 거의 모든 경우에 더 나은 정렬 알고리즘이 존재하기 때문에 **실무에서 버블 정렬이 사용되는 경우는 드뭅니다** [11]. 다만 **거의 정렬된 배열의 경우**에는 O(n)에 가깝게 매우 빠르게 동작하므로, 간단한 문제에서 정렬이 이미 되어 있을 가능 성이 높을 때 써볼 수는 있습니다. 하지만 일반적으로는 삽입 정렬이 거의 정렬된 경우에도 더 효율적이고 구현 도 간단하여, **버블 정렬은 학습용** 이상으로 쓰이는 일은 드물다고 할 수 있습니다 [11]. 실제 라이브러리 구현에 서도 버블 정렬은 찾아보기 어렵고, **Python이나 Java 등의 표준 정렬 라이브러리**는 훨씬 효율적인 퀵/팀/병 합 정렬 등을 사용합니다 [14].

삽입 정렬 (Insertion Sort)

삽입 정렬은 마치 **카드 정렬**을 하듯이, 정렬되지 않은 자료를 하나씩 **이미 정렬된 부분 리스트에 삽입**하여 정렬을 확장해 나가는 알고리즘입니다. 초기에는 첫 원소만으로 정렬된 상태를 유지하고, 이후 두 번째 원소를 알맞은 위치에 삽입, 세 번째 원소를 다시 알맞은 위치에 삽입하는 식으로 진행됩니다. 즉, i번째 단계에서는 앞의 i개 원소는 정렬된 상태로 유지되며, (i+1)번째 원소를 앞쪽의 알맞은 자리에 끼워 넣는 작업을 합니다.

- **동작 과정:** 배열을 **두 부분**으로 나누어 생각합니다. 왼쪽 부분은 **정렬된 부분 배열**, 오른쪽 부분은 **미정렬 부분**입니다. 알고리즘은 미정렬 부분에서 첫 원소를 꺼내 정렬된 부분에 **삽입**하는 것을 반복합니다. 삽입 시, 정렬된 부분의 뒤에서부터 비교하여 적절한 위치를 찾고, 그 위치 이후의 원소들을 한 칸씩 뒤로 밀어 공간을 확보한 뒤 값을 넣습니다. 이렇게 하면 매 단계마다 왼쪽 부분은 계속 정렬 상태를 유지하면서 크기가 1씩 증가합니다.

- **파이썬 구현:** 삽입 정렬은 간결하게 작성할 수 있습니다. 파이썬 리스트를 이용하면 슬라이싱이나 `insert` 메소드를 사용할 수도 있지만, 여기서는 알고리즘 본연의 모습을 보여주기 위해 직접 이동을 구현해 보겠습니다:

```
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
```

```
        j = i - 1
        # arr[j]가 key보다 크면 한 칸 뒤로 밀기
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        # 옳은 위치에 key 삽입
        arr[j + 1] = key

# 사용 예시
data = [5, 2, 4, 6, 1, 3]
insertion_sort(data)
print(data)  # [1, 2, 3, 4, 5, 6]
```

위 코드에서 `i`는 현재 삽입할 원소의 인덱스이며, `key`에 해당 값을 저장해둡니다. 그리고 그 값보다 큰 값들을 정렬된 부분에서 뒤로 한 칸씩 이동시키면서 삽입 위치를 찾습니다. 내부 `while` 루프가 종료되면 `j`가 삽입해야 할 위치 바로 앞을 가리키게 되므로, `arr[j+1]`에 `key`를 놓습니다. 이렇게 하면 `i`번째 반복이 끝났을 때 `arr[:i+1]` 구간은 정렬된 상태가 됩니다.

- **복잡도:** 삽입 정렬의 **시간 복잡도**는 일반적으로 **최악의 경우 O(n^2)**이며, 평균적으로도 O(n^2)입니다. 그러나 **최선의 경우** (이미 배열이 정렬되어 있는 경우)에는 내부 `while`이 한 번도 실행되지 않아서 **O(n)**으로 매우 효율적입니다 [15] [16]. 이는 삽입 정렬이 **부분적으로 정렬된 데이터에 대해 매우 성능이 좋다**는 중요한 특성입니다. 예를 들어 거의 정렬된 배열이나, 새 원소가 추가될 때마다 유지되는 정렬 등에 삽입 정렬을 쓰면 좋은 성능을 기대할 수 있습니다. 실제로 **삽입 정렬은 n이 작을 때도 효율적**이라서, Python의 팀정렬 구현 등에서 **리스트 원소 수가 일정 이하이면 삽입 정렬로 마무리**하기도 합니다. 공간 복잡도는 O(1) 추가 메모리만 사용하므로 **제자리(in-place)** 알고리즘입니다 [17].

- **안정성:** 삽입 정렬은 **안정 정렬**입니다 [17]. 알고리즘상, 동일한 값을 가진 원소의 경우 `arr[j] > key` 조건에서 등호가 없으므로 **앞에 있는 동일값을 뒤로 밀지 않고**, 결국 나중에 삽입되는 동일값은 앞의 동일값 **뒤에 삽입**됩니다. 따라서 입력에서의 상대 순서가 유지됩니다. 위 구현에서도 `>` 비교를 사용하므로 같은 값일 때는 이동을 멈춰, 결과 배열에서 기존 순서를 지켜줍니다. 이 안정성 덕분에 삽입 정렬은 **다중 키 정렬**(secondary sort) 상황에서 유용하게 쓰일 수 있습니다.

- **특징 및 용도:** 삽입 정렬은 **구현이 간단**하고, 특히 **데이터가 거의 정렬되어 있는 경우 매우 빠른** 속도를 냅니다. 예를 들어 길이가 n인 리스트가 이미 정렬되어 있다면, 삽입 정렬은 단 한 번의 비교만으로 각 원소의 위치를 확인하므로 O(n)에 정렬을 완료합니다 [12]. 이 때문에 **실제 프로그램**에서도 완전히 무작위인 데이터보다는 어느 정도 정렬된 데이터가 많다는 점을 이용해, 삽입 정렬이 독립적으로 쓰이기도 하고 또는 복잡한 알고리즘의 보조 루틴으로 활용되기도 합니다 (예: 팀정렬에서 작은 구간을 정렬할 때). 다만 O(n^2)의 최악 복잡도는 여전히 피할 수 없으므로, n이 큰 경우에는 주로 다음에 소개할 O(n log n) 알고리즘들로 대체됩니다.

계수 정렬 (Counting Sort)

다음은 비교 연산을 사용하지 않고 선형 시간에 동작할 수 있는 **계수 정렬**을 다룹니다. **계수 정렬**은 정렬해야 할 값들의 **범위(range)**를 활용하는 알고리즘으로, 일정 범위 내의 정수 키를 가진 항목들을 정렬할 때 효율적입니다. 예를 들어 점수(0점부터 100점 사이)로 구성된 리스트를 정렬할 때, 비교를 일일이 하지 않고 **각 점수가 몇 번 나오는지 세어** 보는 방식으로 정렬할 수 있습니다. 계수 정렬은 이렇게 **각 키 값의 빈도를 세는(counting)** 데서 이름이 유래되었습니다.

- **동작 원리:** 계수 정렬은 크게 세 단계로 이루어집니다. (1) 입력 배열을 한 번 훑으며 **각 키 값의 출현 횟수**를 센다. (2) 그 **누적 개수**를 계산하여 각 키가 정렬된 결과에서 차지할 위치 범위를 계산한다. (3) 입력을 다시 훑으면서 각 원소를 **올바른 위치에 배치**한다. 이때 뒤에서부터 배치하면 안정성을 유지할 수 있습니다. 이러한 과정

을 거치면 비교 연산 없이도 정렬이 가능합니다. 다만 **키 값의 범위**(최댓값 - 최솟값)가 입력 크기에 비해 지나치게 크면 메모리와 시간이 비효율적이므로, 계수 정렬은 **키 범위가 상대적으로 작은 경우에 적합**합니다 [18].

- **파이썬 구현:** 계수 정렬은 별도의 **카운트 배열**과 **출력 배열**을 사용하여 구현합니다. 예를 들어 0 이상인 정수로 구성된 리스트를 정렬하는 코드를 작성하면 다음과 같습니다:

```
def counting_sort(arr):
    if not arr:
        return []  # 빈 리스트 처리
    max_val = max(arr)
    count = [0] * (max_val + 1)
    # 1. 빈도 계산
    for x in arr:
        count[x] += 1
    # 2. 누적 합으로 각 값의 최종 위치 계산
    for i in range(1, len(count)):
        count[i] += count[i - 1]
    # 3. 안정적으로 output 배열에 값 배치
    output = [0] * len(arr)
    for x in reversed(arr):
        count[x] -= 1
        output[count[x]] = x
    return output

# 사용 예시
data = [4, 2, 2, 8, 3, 3, 1]
result = counting_sort(data)
print(result)  # [1, 2, 2, 3, 3, 4, 8]
```

위 구현을 설명하면, 먼저 입력 리스트를 한 바퀴 돌며 `count[x]`에 각 값 `x`의 빈도를 누적합니다. 그런 다음 `count` 배열을 누적합 형태로 변환하는데, 이렇게 하면 특정 값이 **정렬된 출력에서 차지하는 끝 위치**를 나타내게 됩니다. 마지막 단계에서는 입력을 **역순으로 순회**하면서 해당 값을 출력 배열의 올바른 위치에 넣고 `count`를 감소시킵니다. 입력을 역순으로 처리하는 이유는 **안정 정렬**을 위함인데, 먼저 나온 값이 나중에 나오값과 같을 경우 나중 요소부터 배치해야 **원래의 앞뒤 순서**가 유지됩니다. 이 과정을 거치면 결과 배열 `output`에 정렬된 값들이 담겨 반환됩니다.

- **복잡도:** 계수 정렬의 **시간 복잡도**는 배열 길이를 n, 정수 값의 최대 범위를 k라고 할 때 **O(n + k)**입니다 [19]. 배열을 두 번 순회하므로 O(n), 그리고 크기 k인 `count` 배열을 초기화하고 누적합 계산을 하므로 O(k)가 더해집니다. **공간 복잡도** 역시 **O(n + k)**로, 출력용 배열과 카운트용 배열을 추가로 사용합니다 [20]. 이렇듯 계수 정렬은 **입력 크기와 값의 범위에 모두 선형적으로 비례**하므로, k가 n에 비해 충분히 작으면 (예: k = O(n)) 거의 선형 시간에 매우 빠르게 동작합니다. 그러나 k가 n보다 훨씬 크면 비효율적일 수 있습니다. 예를 들어 0부터 10억까지의 숫자가 들어 있을 수 있는 배열 10개를 정렬한다면, 10개의 요소를 정렬하는데 10억 크기의 배열을 사용해야 하므로 오히려 비교식 정렬보다 못할 것입니다.

- **안정성:** 계수 정렬은 일반적으로 **안정 정렬**로 구현됩니다. 위 코드에서도 보이듯이, 동일한 값을 가진 요소들이 입력에서 등장한 **역순으로 출력 배열에 채워지도록** 함으로써 **입력에서의 상대적 순서가 유지**되게 했습니다. 사실 계수 정렬은 **기수 정렬(radix sort)**의 하위 루틴으로 자주 사용되는데, 그 맥락에서 **반드시 안정적이어야만 올바르게 동작**합니다 [21]. 따라서 계수 정렬을 구현할 때 안정성을 염두에 두어야 하며, 불안정하게 구현하면 의도한 결과가 나오지 않을 수 있습니다.

- **특징 및 용도:** 계수 정렬은 **비교 연산을 수행하지 않기 때문에** 입력 값의 분포만 적절하다면 **매우 빠른 선형 시간**에 정렬이 가능합니다 [22]. 예를 들어 최대 점수가 100점인 시험 점수 1만 개를 정렬한다면, n=10000, k=101이므로 O(n+k) = O(10100), 즉 만여 번의 연산으로 정렬이 끝납니다. 이는 O(n log n) 방식으 로는 약 10만 번 이상의 연산이 필요한 것에 비하면 상당히 효율적입니다. 다만, **값이 클수록 (또는 범위가 넓을 수록)** 메모리 사용량과 시간 모두 선형적으로 증가하기 때문에, 키 값 범위가 지나치게 크거나 부동소수점, 문자 열 등 직접 카운트로 처리하기 어려운 데이터에는 사용할 수 없습니다. 현실적으로 계수 정렬은 **정수 집합 정렬** 에 특화되어 있으며, **특정 범위에 국한**된 데이터를 다룰 때 가장 빛을 발합니다. 또한 **기수 정렬**의 내부 과정으 로 활용되어 폭넓은 응용을 가지기도 합니다.

O(n log n) 정렬 알고리즘

이제 대부분의 **실무 환경에서 주로 사용하는 정렬 알고리즘**들인 **O(n log n)** 복잡도의 알고리즘들을 살펴보겠습니 다. 이 범주에는 **병합 정렬**, **퀵 정렬**, **힙 정렬**이 대표적이며, 이들은 **분할 정복**이나 **힙 자료구조** 등을 활용하여 효율적으 로 정렬을 수행합니다. 평균적으로 n의 로그 배 정도의 단계에서 모든 원소를 처리하게 되므로, n이 큰 경우에도 성능이 좋습니다. 다만 각각 **상황에 따른 장단점**이 있으므로, 동작 원리와 특성을 비교해 보는 것이 중요합니다.

병합 정렬 (Merge Sort)

병합 정렬은 **분할 정복(divide and conquer)** 기법을 정렬 문제에 적용한 대표적인 알고리즘입니다. 문제를 절반으로 **분할(divide)**하고, 각각을 재귀적으로 정렬한 다음, 두 정렬된 부분을 **합병(merge)**하여 완성된 결과를 얻는 방식으로 동작합니다. 병합 정렬의 핵심은 두 개의 **정렬된 리스트를 효율적으로 합치는** 병합 단계이며, 이는 비교 기반 정렬이지만 이미 정렬된 구조를 활용하기 때문에 선형 시간에 가능합니다. 전체 알고리즘은 재귀적인 분할 단계 (로그 단계)와 각 단계마다의 선형 병합 작업이 곱해져 **O(n log n)**의 시간 복잡도를 가집니다.

- **동작 과정:** 병합 정렬은 다음과 같이 요약됩니다 [23] [24] :
- **분할(Divide):** 정렬할 배열(또는 리스트)을 두 개의 크기가 거의 같은 부분으로 분할합니다 (전반부와 후반부로 나눔).
- **정복(Conquer): 각 부분 배열을 재귀적으로 병합 정렬**합니다. 부분 배열의 크기가 1이 될 때까지 재귀 호출이 내려갑니다.
- **결합(Combine):** 정렬된 두 부분 배열을 **하나의 정렬된 배열로 병합**합니다. 이때 두 부분이 이미 개별적으로 정렬되어 있으므로 선형 시간에 두 배열을 한 배열로 합칠 수 있습니다.

이 과정은 재귀 호출의 트리로 나타낼 수 있으며, **완전 이진 트리 형태**로 분할이 이루어집니다. 리프 단계에서 길이 1의 배열들이 되고, 병합 과정에서 점진적으로 정렬된 배열로 합쳐집니다.

- **파이썬 구현:** 병합 정렬은 재귀적으로 구현하기 좋습니다. 파이썬에서는 리스트 슬라이싱으로 손쉽게 분할할 수 있으며, 두 리스트의 병합은 투 포인터 방식으로 구현합니다:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr  # 길이 0이나 1이면 이미 정렬 완료
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    # 두 부분을 병합
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
```

```
        if left[i] <= right[j]:
            merged.append(left[i]); i += 1
        else:
            merged.append(right[j]); j += 1
    # 남은 부분 연결
    merged.extend(left[i:]); merged.extend(right[j:])
    return merged

# 사용 예시
data = [38, 27, 43, 3, 9, 82, 10]
print(merge_sort(data))
# [3, 9, 10, 27, 38, 43, 82]
```

위 코드에서 `merge_sort` 함수는 재귀적으로 리스트를 반씩 분할하여 정렬하고, `merged` 리스트를 만들어 두 정렬된 리스트 `left`와 `right`를 병합합니다. 병합 단계에서는 두 리스트의 앞쪽 원소들을 하나씩 비교하여 더 작은 것을 결과에 추가하는 과정을 반복합니다. 하나의 리스트가 끝나면 나머지 리스트의 남은 원소들을 모두 붙여주면 병합 완성입니다. 이 구현에서 병합 시 `<=` 비교를 사용했으므로, **동일한 값일 경우 `left` 리스트의 원소가 우선 배치되어 안정성이 유지**됩니다.

- **복잡도:** 병합 정렬의 **시간 복잡도**는 **최선/평균/최악의 경우 모두 O(n log n)**입니다 [25] . 분할 단계의 깊이 가 $\log_2 n$이고, 각 단계에서 O(n) 시간의 병합이 이뤄지므로 전체 O(n log n)이 됩니다. 이미 정렬 된 경우에도 분할과 병합 과정을 그대로 수행하므로 특별히 더 빨라지지는 않습니다 (일부 구현에서는 병합 전 에 정렬 여부를 검사하여 최적화하기도 하지만, 일반적인 구현은 항상 같은 분할/병합 과정을 거칩니다). **공간 복잡도**는 배열의 경우 **O(n)**의 추가 공간이 필요합니다 [26] . 병합을 위한 임시 배열(혹은 리스트)이 필요하기 때문인데, 위 구현에서도 `merged` 리스트가 입력 크기만큼 할당됩니다. 다만 **연결 리스트로 구현**하면 포인터를 조작하여 병합할 수 있으므로 추가 공간 없이도 (즉 O(1) 공간으로) 병합 정렬을 수행할 수 있습니다 [6] .

- **안정성:** 병합 정렬은 **안정 정렬**입니다. 병합 과정에서 **등값을 만났을 때 앞 리스트(`left`)의 원소를 먼저 결과에 넣는 방식**으로 구현하면, 원래 `left` 쪽 (즉 원래 배열에서 앞쪽)에 있던 원소가 출력에서도 앞에 유지되므로 안정성이 확보됩니다. 위 구현에서도 `<=`를 사용하여 안정적으로 병합했습니다. 단, 배열을 병합할 때는 간단하지만 **링크드 리스트를 병합할 때**는 등값의 순서를 유지하려면 구현에 주의가 필요합니다. 어쨌든, 병합 정렬 자체는 안정하게 구현할 수 있는 알고리즘으로, **C++ 표준 라이브러리의** `stable_sort` 등이 병합 정렬 원리를 활용하고 있습니다 [26] .

- **특징 및 용도:** 병합 정렬은 **예측 가능한 성능**을 가지고 있어, **항상 O(n log n)**에 작업을 완료한다는 것이 큰 장점입니다. 퀵 정렬과 달리 최악의 경우에도 성능이 떨어지지 않으므로, **안정적인 실행 시간**이 요구되거나 최 악 사례가 자주 등장하는 상황에서 유리합니다. 또한 **외부 정렬**(external sorting)에도 흔히 사용되는데, 대용 량 데이터를 부분 정렬한 후 병합하는 형태로 확장될 수 있기 때문입니다. **연결 리스트의 정렬**에도 병합 정렬이 많이 쓰이는데, 배열 기반 정렬들은 임의 접근이 어려운 연결 리스트에 부적합한 반면, 병합 정렬은 노드 연결을 바꾸는 방식으로 쉽게 구현할 수 있고 추가 공간도 거의 필요 없다는 장점이 있습니다 [6] .

반면, 병합 정렬은 배열의 경우 **제자리 정렬이 아니라는 단점**이 있습니다 [26] . 즉, 배열 크기만큼의 임시 공간이 필요하므로 메모리 사용량이 늘어납니다. 또, 재귀 호출을 사용하므로 함수를 호출하는 오버헤드와 스택 메모리도 소량 발생합니다. 그럼에도 불구하고 **안정성**이 요구되거나 **링크드 리스트/외부 정렬 같이 다른 정렬이 곤란한 상황**에서는 병합 정렬이 탁월한 선택입니다.

퀵 정렬 (Quick Sort)

퀵 정렬은 평균적으로 매우 빠른 수행 속도로 인해 실무에서 가장 널리 쓰이는 정렬 알고리즘 중 하나입니다. 퀵 정렬은 이름처럼 **분할과 정복**을 빠르게 해내는 알고리즘으로, 배열을 분할하는 과정 자체를 정렬의 일부로 사용하는 것이 특징

입니다. 구체적으로는 배열에서 하나의 **피벗(pivot)** 값을 선택하고, 그 pivot보다 작은 요소들은 pivot의 왼쪽으로, 큰 요소들은 오른쪽으로 보내 배열을 둘로 **분할(partition)**합니다. 이렇게 하면 pivot의 최종 위치가 확정되며, 이후 pivot을 기준으로 왼쪽 부분과 오른쪽 부분을 **재귀적으로 정렬**하면 전체가 정렬됩니다. 이 과정에서 분할이 균등하게 일어날수록 효율적이며, 평균적으로 매우 뛰어난 성능을 보입니다.

- **동작 과정:** 퀵 정렬은 다음과 같이 요약할 수 있습니다:
- **피벗 선택:** 분할을 위한 기준 요소인 **피벗**을 배열에서 하나 선택합니다. (피벗 선택 전략은 다양하며, 단순히 첫 요소나 마지막 요소를 선택하거나, 중앙값을 추정하는 기법(예: median-of-three) 등을 사용합니다.)
- **분할(Partition):** 배열을 순회하여 피벗보다 작은 값들은 피벗의 왼쪽으로, 큰 값들은 피벗의 오른쪽으로 보내는 **분할 작업**을 수행합니다. 이때 피벗은 최종적으로 두 부분 사이의 위치에 놓이게 되며, 이 위치에 들어간 피벗은 정렬이 완료된 것입니다.
- **재귀 호출:** 피벗을 제외한 **왼쪽 부분과 오른쪽 부분을 재귀적으로 정렬**합니다. 분할된 부분 배열들이 더 이상 분할될 수 없을 때 (부분 배열 크기가 0이나 1일 때) 재귀 호출이 종료됩니다.

이 과정에서 **분할 알고리즘**(partition)이 퀵 정렬의 핵심이며, 제자리로 이 작업을 수행할 수 있습니다. 퀵 정렬은 평균적으로 분할이 균등하게 일어나면 재귀 깊이가 log n이고 매 분할당 O(n) 작업으로 **평균 O(n log n)**의 복잡 도를 보입니다.

- **파이썬 구현:** 파이썬으로 퀵 정렬을 구현할 때, 리스트 편의를 활용하면 비교적 짧게 작성할 수도 있습니다 (리스트 컴프리헨션을 이용한 비제자리 구현). 그러나 여기서는 **제자리(in-place)** 분할을 수행하는 형태로 구현해 보겠습니다 (호어(Hoare) 분할 등 표준 기법 중 Lomuto 파티셔닝을 사용):

```
def quick_sort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1
    if low < high:
        # Lomuto partition: 마지막 원소를 피벗으로 선택
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        # pivot을 올바른 위치에 놓기
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        p = i + 1  # pivot의 최종 위치
        # 분할된 두 부분을 재귀적으로 정렬
        quick_sort(arr, low, p - 1)
        quick_sort(arr, p + 1, high)

# 사용 예시
data = [10, 7, 8, 9, 1, 5]
quick_sort(data)
print(data)  # [1, 5, 7, 8, 9, 10]
```

위 구현에서는 항상 마지막 원소 `arr[high]`를 피벗으로 선택합니다. `for` 루프를 통해 피벗보다 작거나 같은 원소들을 배열 앞부분으로 모읍니다. 포인터 `i`는 현재까지 확인한 영역에서 피벗보다 작거나 같은 원소들의 끝 위치를 가리키며, `j`가 움직이며 해당 조건을 만족하면 `i`를 증가시키고 교환합니다. 루프가 끝나면 `i` 인덱스 앞에는 피벗

보다 작거나 같은 값들이, `i+1`부터 `high-1`까지는 피벗보다 큰 값들이 위치하게 됩니다. 마지막으로 피벗을 `i+1` 위치로 교환하면 분할 완료됩니다. 그 후 재귀 호출로 왼쪽 부분과 오른쪽 부분을 각각 정렬합니다. (참고: 여기서는 `<=` 조건을 사용하여 피벗과 동일한 값은 피벗의 왼쪽 영역으로 보내고 있습니다. 이렇게 하면 동일한 값들이 왼쪽 영역에 몰리게 되는데, 이는 피벗과 동일한 값의 상대적 순서에는 영향을 주지 않지만 **퀵 정렬 자체는 불안정**하므로 큰 의미는 없습니다.)

- **복잡도:** 퀵 정렬의 **시간 복잡도**는 **평균적으로 O(n log n)**, **최선의 경우 O(n log n)**, **최악의 경우 O(n^2)**입니다 [27] [28]. 평균과 최선의 상황은 분할이 균형 있게 이루어질 때이며, 최악은 분할이 극단적으 로 한쪽으로 치우칠 때입니다. 예를 들어 피벗을 항상 최솟값이나 최댓값으로 선택하게 되면 한쪽 부분은 빈 배 열이 되고, 다른 쪽은 원래보다 1만 줄어든 배열이 되어 (거의 정렬된 입력에서 이런 경우가 발생) 재귀 깊이가 n에 달해 O(n^2)이 됩니다. 다행히도 평균적으로는 피벗이 random하게 선택된다고 가정할 때 O(n log n)에 가까 운 분할 결과가 나오며, **퀵 정렬은 평균적으로 대부분 정렬 알고리즘 중 가장 빠른 축**에 속합니다 [29]. 또한 **실제 데이터에서 캐시 효율 등 구현상의 이점**도 커서, 정렬 라이브러리의 기본 구현으로 채택되는 경우가 많습니다 [29]. **공간 복잡도**는 제자리 구현의 경우 추가 배열은 사용하지 않지만, 재귀 호출 스택으로 **O(log n)**(평균) 공간 을 사용합니다 [30]. 최악의 경우 재귀 깊이가 O(n)까지 늘어날 수 있지만, 이러한 상황을 막기 위해 대부분의 구현은 재귀 깊이가 일정 수준 이상 깊어지면 다른 알고리즘으로 전환하는 등의 조치를 합니다.

- **안정성: 퀵 정렬은 불안정 정렬**입니다. 분할 과정에서 요소들을 교환하기 때문에, 동일한 값의 상대 순서가 보존되지 않습니다 [28]. 예를 들어 `[▸5, 5▸, 1]` 배열에서 피벗 1을 기준으로 5들과 교환이 일어나거나, 혹은 피벗 5를 기준으로 같은 값 5와의 불필요한 교환 등이 발생하면 결과에서 순서가 바뀔 수 있습니다. 퀵 정렬을 안정적으로 구현하는 방법도 연구되어 있지만 (예: 불안정성을 없애기 위해 추가 메모리에 요소의 원래 인덱스를 보관하는 등), 일반적으로 퀵 정렬은 안정성이 요구되지 않는 상황에서 사용됩니다.

- **특징 및 용도:** 퀵 정렬은 **대부분의 경우 가장 빠른 정렬 알고리즘**으로 알려져 있습니다. 평균적으로 $2 \times n \log_2 n$ 정도의 비교를 수행하며 [28], 상수 계수가 작아 같은 O(n log n)이라도 병합 정렬이나 힙 정렬보다 빠르게 동작하는 경향이 있습니다. 다만, 최악의 경우가 O(n^2)이라는 점은 이론적 약점이며, 이를 보완하기 위해 **무작위 피벗 선택**(randomized quicksort, 최악 확률을 낮춤)이나 **median-of-three** 등 피벗 선 택을 개선하는 기법이 사용됩니다. 더욱 철저하게는, **C++ 표준 정렬(Introsort)**처럼 **퀵 정렬 수행 중 재귀 깊이가 깊어져 최악 상황이 될 것 같으면 힙 정렬로 갈아타는** 전략도 있습니다 [31]. 이러한 변종들을 통해 사실상 퀵 정렬의 최악 성능 문제는 대부분 해결됩니다.

퀵 정렬은 불안정하지만, **안정성이 필요 없는 대부분의 일반 목적 정렬에 적합**합니다. 또한 **제자리 정렬**로 추가 메모리가 거의 들지 않고, **재귀 호출을 제외하면 구현이 비교적 간결**한 편입니다. (다만 재귀 호출은 컴퓨터 시스템의 **스택 오버플로우** 위험을 내포하므로, 매우 큰 n에 대해서는 반복구현이나 Tail Call Optimization 등이 고려되기도 합니다.) 총합적으로, 퀵 정렬은 **속도**, **공간 효율** 측면에서 뛰어나 **많은 언어/라이브러리에서 기본 정렬 알고리즘**으로 채택되어 있습니다. (파이썬은 객체 정렬에 팀정렬을 쓰지만, C++의 `std::sort`는 퀵 정렬 기반의 인트로소트를 사용합니다.)

힙 정렬 (Heap Sort)

힙 정렬은 **힙(heap)** 자료구조를 이용하여 정렬을 수행하는 알고리즘입니다. 최대 힙(max-heap)을 활용하면 가장 큰 원소를 O(1)에 얻을 수 있다는 점에 착안하여, 입력 배열을 최대 힙으로 만든 후 **가장 큰 원소부터 차례로 제거하여** 정렬을 완성합니다. 구체적으로, 배열을 최대 힙으로 변환하면 루트(인덱스 0)가 최댓값이 되고, 이를 마지막 원소와 교환해 배열의 끝에 배치합니다. 힙 크기를 1 줄인 후 다시 루트 자리에 떨어진 원소를 힙으로 조정(heapify)하면 남은 부분에 대해 다시 최댓값이 루트에 위치합니다. 이 과정을 반복하면 큰 값부터 끝쪽으로 채워지며 오름차순 정렬이 이루어집니다. 힙 정렬의 특징은 **최악의 경우에도 O(n log n)**을 보장하며, **제자리 정렬**이라는 점입니다.

- **동작 과정:** 힙 정렬은 두 단계로 나눌 수 있습니다:

- **힙 구축(Build Heap):** 주어진 배열을 최대 힙 구조로 변환합니다. (전체 배열을 한 번의 과정을 통해 힙으로 만들 수 있으며, 이를 **heapify 과정**이라고 합니다. 시간 복잡도 O(n)에 수행 가능합니다.)

- **반복적 제거 및 복구:** 배열의 마지막 인덱스부터 역순으로 채워나가는 개념으로, 현재 힙의 루트(최댓값)를 **배열의 뒷부분으로 보내고** 힙 크기를 줄입니다. 그런 다음 줄어든 힙에 대해 **힙 속성 복구**(재정렬)를 수행하여 남은 부분에서 다시 최댓값을 루트로 만듭니다. 이 과정을 힙에 원소가 하나도 남지 않을 때까지 반복합니다.

결과적으로 큰 값들이 배열 끝쪽부터 차례대로 쌓이며 정렬이 완료됩니다.

- **파이썬 구현:** 파이썬에서는 내장 모듈 `heapq`가 최소 힙(min-heap)만을 제공하므로, 알고리즘을 명시적으로 구현해 보겠습니다. (코드가 다소 길지만, 개념 이해를 위해 직접 heapify 함수를 작성합니다.)

```python
def heap_sort(arr):
    n = len(arr)
    # 1. 최대 힙 구축 (heapify 과정)
    for i in range(n // 2 - 1, -1, -1):
        heapify_down(arr, i, n)
    # 2. 힙에서 최댓값을 끝으로 보내고 힙 크기를 줄이며 정렬
    for end in range(n - 1, 0, -1):
        arr[0], arr[end] = arr[end], arr[0]   # 루트(최댓값)를 끝으로 보냄
        heapify_down(arr, 0, end)             # 줄어든 힙 크기(end)까지 힙 복구

def heapify_down(arr, i, n):
    # 노드 i를 루트로 하는 부분트리가 힙 특성을 만족하도록 아래로 내려가는 함수
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify_down(arr, largest, n)

# 사용 예시
data = [4, 10, 3, 5, 1]
heap_sort(data)
print(data)   # [1, 3, 4, 5, 10]
```

먼저 `heap_sort` 함수에서 배열의 중간 인덱스부터 역순으로 `heapify_down`을 호출하여 전체 배열을 최대 힙으로 만듭니다. (`n//2 - 1`은 마지막 내부노드의 인덱스입니다. 그 이후의 인덱스들은 리프 노드이므로 내려갈 필요가 없습니다.) `heapify_down` 함수는 노드 `i`의 두 자식과 값을 비교하여 가장 큰 자식과 자리를 바꾸고, 자리 바뀐 자식 노드를 대상으로 재귀적으로 동일 작업을 수행합니다. 이로써 `i`를 루트로 하는 부분 트리가 힙 특성을 만족하게 됩니다.

힙이 구축되면, 이제 가장 큰 값이 `arr[0]`에 있으므로 이를 `arr[end]`와 교환하여 정렬된 위치로 보냅니다. 그런 다음 `arr[0]`에 내려온 원소에 대해 힙 복구를 해주면 남은 부분이 다시 힙이 됩니다. 이 과정을 `end`를 1까지 줄여 가며 반복하면, 배열이 오름차순으로 정렬됩니다.

- **복잡도:** 힙 정렬의 **시간 복잡도**는 **최선/평균/최악 모두 O(n log n)**입니다 [32] . 힙 구축 단계가 O(n), 이 후 n-1번의 반복 과정에서 매번 힙 복구에 O(log n)이 걸리므로, 정확히 O(n + n log n) = O(n log n) 이 됩니다. 특히 입력 데이터 분포에 관계없이 항상 동일한 패턴으로 수행되므로 퀵 정렬처럼 최악의 경우 성 능이 나빠지는 일도 없습니다. **공간 복잡도**는 배열 자체를 힙으로 변환하여 사용하므로 **추가 배열 공간이 불필요(O(1))**하며, 제자리 정렬 알고리즘입니다. (재귀를 사용했다면 O(log n) 스택 공간이 있겠지만, 위 구현은 반복문과 재귀 혼합 형태이며 재귀 깊이가 log n 수준이라 큰 문제는 없습니다.)

- **안정성:** 힙 정렬은 **불안정 정렬**입니다. 힙의 구성 및 요소 교환 과정에서 동일한 값의 상대적인 순서가 보존되지 않습니다 [33] . 예를 들어 `[▸2, 1, 2▸]`를 힙 정렬하면, 처음 힙 구축 후 `[2, 1, 2]`(최대 힙)에서 2와 2의 순서는 이미 뒤바뀐 상태가 될 수 있습니다. 따라서 안정성이 필요하다면 힙 정렬은 사용하기 어렵습니다.

- **특징 및 용도:** 힙 정렬의 가장 큰 강점은 **최악의 경우에도 O(n log n)을 보장**한다는 점과, **추가 메모리를 거의 사용하지 않는 제자리 정렬**이라는 점입니다 [32] . 이러한 이유로 퀵 정렬의 최악 성능을 꺼리는 환경에서 고려할 만합니다. 예를 들어, **시간 복잡도가 중요하게 요구되는 실시간 시스템**이나 **보안 측면에서 입력에 따른 성능 편차가 크면 곤란한 경우** 등에 힙 정렬을 채택할 수 있습니다. 또한 최대값/최소값을 반복적으로 뽑아내는 **선택 문제**(selection problem)나 **우선순위 큐** 구현 등과 개념을 공유하기 때문에, 정렬 이외의 상황에서도 힙 자료구조와 함께 활용됩니다.

그러나 **실제로 정렬 라이브러리에서 힙 정렬이 채택되는 경우는 드뭅니다**. 이유는 **평균 속도가 퀵 정렬보다 느린 경향**이 있고, 데이터 접근 패턴이 뛰어나지 않아 캐시 효율이 떨어지기 때문입니다. 대신 C++의 `std::sort`처럼 **인트로소트(Introsort)** 전략으로 **일반적으로는 퀵 정렬을 쓰되, 최악 상황이 될 것 같으면 힙 정렬로 전환**하는 방식이 활용됩니다 [31] . 요컨대 힙 정렬은 **일관된 O(n log n) 성능과 낮은 공간 오버헤드**가 장점이지만, **평균적인 실제 속도는 다소 떨어지는** 알고리즘입니다.

기수 정렬 (Radix Sort)

기수 정렬은 숫자나 문자열 등의 **자릿수(digit) 정보를 활용**하여 정렬하는 **비교 연산을 사용하지 않는** 알고리즘입니다. 여러 개의 키를 가진 데이터를 정렬할 때, **하위 키부터 차례로 정렬**해나가는 방식으로 작동합니다. 예를 들어 10진수 정수들을 정렬한다고 하면, 일의 자리부터 정렬하고, 그 결과를 유지한 채 십의 자리로 정렬하고, 계속 진행하여 가장 높은 자리까지 정렬하는 식입니다. 이렇게 하면 최종적으로 전체 숫자가 정렬됩니다. 각 자릿수의 정렬에는 **계수 정렬**과 같은 안정적인 알고리즘을 사용하여 자릿수별 정렬 결과가 상위 자릿수 정렬에서 유지되도록 합니다.

- **동작 과정:** 기수 정렬에는 크게 **LSD (Least Significant Digit) 방식**과 **MSD (Most Significant Digit) 방식**이 있습니다. LSD 방식은 위에서 언급한 대로 **가장 낮은 자리부터** 정렬해 올라가는 방식이고, MSD 방식은 반대로 **가장 높은 자리부터** 정렬하고 필요하면 그룹을 세분화하여 재귀적으로 진행하는 방식입니다. 일반적으로 숫자 정렬에서는 LSD 방식을 많이 사용하고, 문자열(가변 길이)의 경우 MSD 방식을 사용하기도 합니다.

LSD 기수 정렬의 예를 간단히 들면 다음과 같습니다. 다음 리스트를 생각해봅시다: `[170, 45, 75, 90, 802, 24, 2, 66]`. LSD 방식으로 진행하면: 1. **1의 자리 정렬:** 일의 자리만 보고 정렬하면 -> `[170, 90, 802, 2, 24, 45, 75, 66]` (1의 자리 오름차순) 2. **10의 자리 정렬:** (1의 자리가 같은 경우 이전 순서 유지) 10의 자리로 정렬하면 -> `[802, 2, 24, 45, 66, 170, 75, 90]` 3. **100의 자리 정렬:** -> `[2, 24, 45, 66, 75, 90, 170, 802]` (최종 정렬 결과)

여기서 각 단계는 안정적인 방법으로 수행되어, 상위 자리의 순서가 바뀌지 않도록 합니다 [34]. 이 과정에서 2자리 수나 1자리 수에는 필요한 만큼 앞에 0을 붙여 (`002`, `024` 등) 생각합니다.

- **파이썬 구현:** 일반적인 10진수 양의 정수를 정렬하는 LSD 기수 정렬을 구현해 보겠습니다. (코드에서는 매 자릿수를 계수 정렬하는 방식을 사용합니다.)

```python
def radix_sort(arr):
    if not arr:
        return arr
    max_val = max(arr)
    exp = 1  # 10^0, 10^1, ... 자릿수를 나타내는 exponent
    output = [0] * len(arr)
    while max_val // exp > 0:
        # 계수 정렬을 현재 자릿수(exp)에 대해 수행
        count = [0] * 10
        for x in arr:
            digit = (x // exp) % 10
            count[digit] += 1
        for i in range(1, 10):
            count[i] += count[i - 1]
        # 안정성을 위해 입력을 역순으로 훑으며 output에 배치
        for x in reversed(arr):
            digit = (x // exp) % 10
            count[digit] -= 1
            output[count[digit]] = x
        # 다음 자릿수 처리를 위해 arr 업데이트
        arr = output[:]
        exp *= 10
    return arr

# 사용 예시
data = [170, 45, 75, 90, 802, 24, 2, 66]
print(radix_sort(data))
# [2, 24, 45, 66, 75, 90, 170, 802]
```

코드 설명: `exp`는 현재 처리할 자릿수의 단위를 나타내며, 1부터 10, 100, ...으로 증가합니다. 각 루프에서는 `count` 배열을 사용하여 해당 자릿수의 숫자 빈도를 세고, 누적합을 구하여 각 숫자가 출력 배열에서 차지할 최종 위치를 계산합니다. 그런 다음 입력 배열을 역순으로 훑으며 (안정 정렬을 위해) 해당 자릿수의 값에 따라 `output` 배열의 정확한 위치에 원소를 넣습니다. 이렇게 한 자릿수에 대한 정렬이 끝나면 `arr`를 정렬된 `output`으로 업데이트하고 다음 자릿수로 넘어갑니다. `max_val // exp > 0`인 동안 (즉 현재 자릿수가 최대 자릿수보다 작은 동안) 계속 반복합니다.

- **복잡도:** 기수 정렬의 **시간 복잡도**는 각 자릿수 정렬에 걸리는 시간에 자릿수 개수를 곱한 **O(d × (n + k))**로 흔히 표현됩니다 [35]. 여기서 d는 자릿수 개수, k는 각 자릿수의 가능한 값의 수입니다 (10진수이면 k=10). 보통 10진수 숫자를 다룰 때 k는 상수 10이므로 시간 복잡도를 **O(d × n)**으로 볼 수 있습니다. d는 정수의 자릿수 길이에 비례하며, 만약 최대값이 M이라면 d = O(log_10 M)입니다. 따라서 전체 복잡도를 입력 크기 n과 값의 크기 M으로 표현하면 **O(n × log M)** 정도라고 할 수 있습니다. 실제로 많은 경우 32비트 정수라면 d가 최대 10 (10자리수) 정도로 작기 때문에, 기수 정렬은 사실상 **선형 시간**에 가까운 성능을 냅니다 [36]. **공간 복잡도**는 계수정렬과 마찬가지로 출력 배열과 카운트 배열을 사용하므로 **O(n + k)**입니다. 10진수의 경우 k=10으로 미미하므 로 **실질적으로 O(n)의 추가 공간**을 사용 하는 셈입니다.

- **안정성:** 기수 정렬은 **안정 정렬**이어야만 올바르게 동작합니다 [37] . 각 자릿수를 정렬할 때 안정적인 방법(예: 계 수 정렬의 안정적인 구현)을 사용해야 상위 자릿수의 순서가 유지됩니다. 위 구현에서도 내부적으로 안정성을 지키도록 뒷부분부터 배열을 훑으며 출력 배열에 배치했습니다. 만약 불안정하게 정렬하면, 하위 자릿수 기준 정렬 결과가 깨져버려 최종 결과가 엉망이 될 것입니다. 기수정렬은 그 **동작 원리상 안정성이 필수**라는 점을 기 억해야 합니다 [38] .

- **특징 및 용도:** 기수 정렬은 비교 기반 정렬의 이론적 하한을 깨고 **선형에 가까운 시간에 정렬을 수행**할 수 있다는 점에서 매력적인 알고리즘입니다. 특히 **정렬할 키가 자릿수로 분해될 수 있고, 자릿수의 개수가 상대적으로 작을 때** 효과적입니다. 예를 들어 자릿수가 6개 이하인 주민등록번호 1억 개를 정렬하는 경우, 기수 정렬은 매우 빠르게 동작할 것입니다. 문자열 정렬의 경우에도 각 문자를 자리로 보고 LSD 또는 MSD 기법을 적용할 수 있습니다. 다만, 기수 정렬은 키 길이가 길어지면 (예: 100자리 정수) d가 커져서 비교식 정렬에 비해 이득이 적어질 수 있습니다. 또한 **부동소수점이나 가변길이 문자열**처럼 직접적인 자릿수 분리가 까다로운 데이터에도 적용 하기 어렵습니다.

정렬할 데이터가 32비트 정수와 같이 **고정된 크기의 키**를 가지고 있다면, 많은 경우 기수 정렬이 n log n 알고리즘들 보다 빠르게 동작할 수 있습니다. 하지만 구현이 상대적으로 복잡하고, 추가 메모리를 사용하며, 키 추출 등의 오버헤드 가 존재하 므로 데이터 크기나 환경에 따라 실제 성능은 달라질 수 있습니다. 현대의 정렬 라이브러리들은 대부분 퀵/병 합/힙 정렬 기반이지만, 상황에 따라 **특수한 경우 기수 정렬을 활용**하는 경우도 있습니다 (예: 특정 경우에 **팀정렬**이 삽 입 정렬을 쓰듯, 큰 정수를 정렬할 때 기수 정렬로 최적화하는 등).

고급 정렬 기법 소개

앞서 다룬 정렬 알고리즘 이외에도 다양한 응용 및 개선 기법이 존재합니다. 여기서는 추가로 알아두면 좋은 몇 가지 **고급 정렬 알고리즘과 기법**을 간략히 소개합니다 (자세한 구현은 생략).

- **셸 정렬 (Shell Sort):** 셸 정렬은 삽입 정렬을 일반화하여 **멀리 떨어진 요소들끼리도 미리 교환**하는 기법입니다. 일정한 **간격(gap)**으로 떨어진 요소들을 그룹으로 묶어 부분 삽입 정렬을 수행하고, gap을 점차 줄여가면서 최종적으로 gap=1일 때 삽입 정렬을 완료합니다. 이렇게 하면 **멀리 있는 요소가 조금씩 제자리로 이동**하여 삽입 정렬의 단점을 개선합니다. 셸 정렬의 성능은 사용되는 gap 수열에 따라 달라지며, 이론적으로 **시간 복잡도가 완전히 규명되지 않은 알고리즘**입니다 [39] . 여러 연구 결과 실험적으로는 O(n^{1.3}) 내외의 성능을 보이는 gap도 있고 최악 O(n^2)인 수열도 있습니다. **안정성**은 삽입 정렬과 달리 **보장되지 않습니다** [40] (큰 gap으로 뛰어넘으며 삽입하므로 동일 값 순서가 뒤바뀔 수 있음). 셸 정렬은 구현이 간단하고, n이 작거나 데이터가 부분적으로 정렬된 경우 삽입 정렬보다 빠를 수 있어 과거에 자주 언급되었으나, 최근에는 팀정렬 등의 등장으로 사용 빈도가 높지는 않습니다.

- **버킷 정렬 (Bucket Sort):** 계수 정렬과 유사한 아이디어로, 데이터를 미리 몇 개의 **버킷(bucket)**으로 나누어 담은 다음 각 버킷을 개별 정렬하고 합치는 알고리즘입니다. 데이터가 **균일하게 분포**되어 있으면 각 버킷에 거의 균등하게 데이터가 담겨 효율적입니다. 예를 들어 [0, 1) 범위의 실수 1백만 개를 정렬한다고 할 때 100개의 버킷으로 분산하고 각 버킷을 정렬하면 빠릅니다. **평균 시간 복잡도**는 분포 가정 하에 **O(n + m)** 정도가 되며 (m은 버킷 수 및 각 버킷 정렬 비용) [41] , **최악의 경우** 모든 데이터가 하나의 버킷에 들어가면 **O(n^2)**까지도 떨어질 수 있습니다 [42] . 따라서 입력 분포를 어느 정도 알고 있을 때 유용한 기법입니다. 버킷 정렬은 내부적으로 어떤 정렬을 쓰느냐에 따라 안정성 여부가 갈리지만, 보통 **삽입 정렬 같은 안정 정렬을 버킷별로 적용 하면 안정적**입니다. 주로 균일 분포를 가정할 수 있는 **특정 확률적 시뮬레이션**이나 **그래픽스** 분야 등에서 사용 되는 경우가 있습니다.

- **팀정렬 (Timsort):** 팀정렬은 2002년에 파이썬의 리스트 정렬을 위해 고안된 알고리즘으로, **삽입 정렬과 병합 정렬의 하이브리드**입니다. 리스트 내에서 이미 정렬된 **run** 구간들을 찾아낸 뒤, 그 구간들을 병합 정렬하는 방식으로 동작합니다. 또한 일정 크기 이하의 리스트는 삽입 정렬로 처리하는 등 실용적인 최적화들이 포함되어 있습니다. 팀정렬의 성능은 **최악 O(n log n)**, **최선 O(n)** (이미 대부분 정렬된 경우)으로, **데이터가 가진 기존의 정렬된 구조를 최대한 활용**하는 **적응형 정렬**입니다 [43] . **안정 정렬**이기도 하여 객체 정렬 등에서도 유용합니다 [44] . 이러한 이유로 팀정렬은 현재 **Python의 기본 정렬 알고리즘**으로 사용되며 (Java의 `Arrays.sort(Object[])` 등도 팀정렬 기반임), **실제 데이터에서 매우 우수한 성능**을 보이는 것으로 평

가됩니다. 다만 구현이 상당히 복잡한 편이며, 개념적으로 병합 정렬 기반이라 **공간 복잡도 O(n)**를 필요로 합니다.

- **인트로소트 (Introsort):** 인트로소트는 **퀵 정렬의 평균적인 효율성과 힙 정렬의 최악 성능 보장을 결합**한 하이브리드 알고리즘입니다. 1997년 Musser가 고안하여 C++ 표준 정렬에 채택했으며, 동작 원리는 간단합니다: 퀵 정렬을 수행하되, **재귀 깊이가 일정 수준($\log_2 n * 2$ 등)보다 깊어지면 더 이상 퀵 정렬을 하지 않고 힙 정렬로 전환**하는 것입니다 [31]. 이렇게 하면 퀵 정렬의 최악 O(n^2) 상황을 피하고 **항상 O(n log n)**에 종료시킬 수 있습니다. 또한 작은 부분에 대해서는 삽입 정렬로 처리하여 최적화를 도모합니다. 인트로소트는 퀵/힙/삽입 정렬의 좋은 부분들을 조합한 알고리즘으로, **평균 성능은 퀵 정렬과 동등하면서도 최악 성능은 힙 정렬처럼 보장**되는 장점이 있습니다. C++의 `std::sort`가 이 방식을 택하고 있으며, 이는 **불안정 정렬**이지만 **제자리 정렬**이라 추가 공간을 쓰지 않습니다 [45].

이 외에도 **분산 정렬(distribution sort)**, **외부 정렬**, **망갑 정렬(Cocktail sort)**, **Cycle sort** 등 특수한 목적이나 조건에서 사용되는 정렬 알고리즘들이 다수 존재합니다. 그러나 일반적인 코딩 테스트나 실무에서 자주 만나는 경우는 위에서 다룬 알고리즘들이 대부분이며, 그 중에서도 **퀵 정렬, 병합 정렬, 힙 정렬, 그리고 삽입 정렬** 정도를 확실히 이해하고 구현할 수 있으면 충분합니다. 파이썬 등 고수준 언어에서는 이미 최적화된 정렬 라이브러리를 제공하지만, 내부 동작 원리를 이해하는 것은 유용한 지적 도구가 될 것입니다.

정렬 알고리즘 요약 비교

마지막으로, 지금까지 살펴본 주요 정렬 알고리즘들의 특성을 표로 정리합니다:

알고리즘	안정성	시간 복잡도(최선/평균/최악)	공간 복잡도	제자리?	특징/비교
선택정렬	불안정	$\Omega(n^2) / \Theta(n^2) / O(n^2)$	O(1)	예	구현이 가장 간단, 교환 횟수 최소지만 항상 느림 [8].
버블정렬	안정	$\Omega(n) / \Theta(n^2) / O(n^2)$	O(1)	예	거의 정렬된 경우 매우 빠름(adaptive). 교육용, 실무 사용 드묾. [12]
삽입정렬	안정	$\Omega(n) / \Theta(n^2) / O(n^2)$	O(1)	예	대부분의 입력에 빠르고, 거의 정렬된 경우 최적 [15]. 작은 배열 정렬 등에 활용
계수정렬	안정	$\Omega(n + k) / \Theta(n + k) / O(n + k)$	O(n + k)	아니오	키 값이 정수이고 범위 k가 작을 때 유용. 비교 연산 없음 [22].
병합정렬	안정	$\Omega(n \log n) / \Theta(n \log n) / O(n \log n)$	O(n)	아니오	항상 O(nlog n)으로 예측 가능, 연결 리스트 등에 적합 [6]. 추가 메모리 필요.
퀵정렬	불안정	$\Omega(n \log n) / \Theta(n \log n) / O(n^2)$	O(logn)	예	평균적으로 가장 빠른 정렬 [29]. 최악 대비해 피벗 선택 중요. 라이브러리 구현 다수.
힙정렬	불안정	$\Omega(n \log n) / \Theta(n \log n) / O(n \log n)$	O(1)	예	항상 O(nlog n) 보장 [32], 추가 메모리 불필요. 실사용은 인트로소트 등에 포함되어 활용.
기수정렬	안정	$\Omega(d \times n) / \Theta(d \times n) / O(d \times n)$	O(n + k)	아니오	정수, 문자열 등에서 자릿수(d)가 작으면 매우 빠름. 비교식 하한 돌파 [36].

알고리즘	안정성	시간 복잡도(최선/평균/최악)	공간 복잡도	제자리?	특징/비교
셸정렬	불안정	Ω(n log n) / ~Θ(n^1.3) (수열에 따라 다름) / O(n²)	O(1)	예	삽입 정렬 개선판. 성능 분석 어려움, 부분적으로 정렬된 경우 효율적.
버킷정렬	구현에 따름	Ω(n + m) / Θ(n + m + n²/m) / O(n²)	O(n+m)	구현에 따름	분포를 알 때 효율적. 버킷 수 m에 따라 조절, 보통 안정적 정렬 사용.
팀정렬	안정	Ω(n) / Θ(n log n) / O(n log n)	O(n)	아니오	실용 하이브리드: 파이썬 등 기본 정렬. 부분 정렬 활용하여 실제 매우 빠름[43].
인트로소트	불안정	Ω(n log n) / Θ(n log n) / O(n log n)	O(1)	예	퀵+힙+삽입 하이브리드. C++ `std::sort` 등에 사용[31]. 최악 대비책으로 힙 정렬 사용.

(참고: 상기 복잡도의 최선은 특별한 언급이 없는 한 입력이 이미 정렬되어 있는 경우를 가정합니다. 기수 정렬의 d는 키의 자릿수, k는 가능한 키 값의 수이며, 예를 들어 32비트 양의 정수라면 $d \approx 10$, $k = 10$ (10진수 기준)로 상수 취급 가능합니다.)

다음 주 미리보기

정렬 알고리즘에 대한 학습은 여기까지입니다. **다음 6주차에는** 알고리즘 문제 풀이에서 자주 등장하는 **투 포인터(two pointers)** 기법, **슬라이딩 윈도우(sliding window)** 기법, 그리고 **이분 탐색(binary search)** 및 이를 활용한 **매개변수 탐색(parametric search)**, **이진 검색 트리(BST)** 등을 개념 위주로 다룰 예정입니다. 이번 주에 배운 정렬과 자료구조 지식을 바탕으로, 다음 주에는 투 포인터/슬라이딩 윈도우로 **효율적인 탐색**을, 이분 탐색과 매개 변수 탐색으로 **로그arithmic한 문제 해결**을, BST로 **동적 정렬 자료구조** 활용을 배우게 될 것입니다. 수고하셨습니다! 다음 시간에 만나요.

---

[1] [2] [3] [4] [5] Sorting algorithm - Wikipedia
https://en.wikipedia.org/wiki/Sorting_algorithm

[6] [25] [26] Merge sort - Wikipedia
https://en.wikipedia.org/wiki/Merge_sort

[7] [8] [9] [10] Selection sort - Wikipedia
https://en.wikipedia.org/wiki/Selection_sort

[11] [12] [13] [14] Bubble sort - Wikipedia
https://en.wikipedia.org/wiki/Bubble_sort

[15] [16] [17] Insertion sort - Wikipedia
https://en.wikipedia.org/wiki/Insertion_sort

[18] [19] [20] [21] [22] Counting sort - Wikipedia
https://en.wikipedia.org/wiki/Counting_sort

[23] [24] study-note-week04.pdf
file://file-FMJZSbhxQTAoHKJP48s2dT

[27] [28] [29] [30] Quicksort - Wikipedia
https://en.wikipedia.org/wiki/Quicksort

[31] [45] Introsort - Wikipedia
https://en.wikipedia.org/wiki/Introsort

[32] [33] Heapsort - Wikipedia
https://en.wikipedia.org/wiki/Heapsort

34 36 Radix sort - Wikipedia
https://en.wikipedia.org/wiki/Radix_sort

35 Computational Complexity of Radix Sort
https://www.numberanalytics.com/blog/computational-complexity-radix-sort

37 DSA Radix Sort - W3Schools
https://www.w3schools.com/dsa/dsa_algo_radixsort.php

38 Time and Space complexity of Radix Sort Algorithm | GeeksforGeeks
https://www.geeksforgeeks.org/time-and-space-complexity-of-radix-sort-algorithm/

39 40 Shellsort - Wikipedia
https://en.wikipedia.org/wiki/Shellsort

41 42 Bucket sort - Wikipedia
https://en.wikipedia.org/wiki/Bucket_sort

43 Timsort - Wikipedia
https://en.wikipedia.org/wiki/Timsort

44 Timsort in Depth: Analysis and Comparison - Number Analytics
https://www.numberanalytics.com/blog/timsort-in-depth-analysis-and-comparison