계수 정렬 (Counting Sort)

Comparison Sort (비교 정렬)

> N개의 원소의 배열을 정렬하는 가짓수는 N!
> 따라서, Comparison Sort를 통해 생기는 트리의 말단 노드가 N!이상의 노드 갯수를 갖기 위해선 2^h ≥ N!를 만족하는 h(트리의 높이 == 비교정렬의 시간 복잡도)를 가져야 하고, 이 식을 h > O(n logn)을 가져야한다.
>
> O(n logn)을 줄일 수 있는 방법은 Comparison(비교)를 하지 않는 것.

Counting Sort 과정

시간 복잡도 : O(n + k) → k는 배열에서 등장하는 최대값

공간 복잡도 : O(k) → k만큼의 배열을 만들어야 함

Counting이 필요 : 원소의 등장횟수를 센다

- 사용 : 정렬하는 숫자가 특정한 범위 내에 있을 때 사용

  (Suffix Array를 얻을 때, 시간복잡도 O(n logn)으로 얻을 수 있음.)
- 장점 : O(n)의 시간복잡도
- 단점 : 배열 사이즈 N만큼 돌 때, 증가시켜주는 Counting 배열의 크기가 크게 만들어야 한다.
  - 1 ~ 5를 담을 Counting 배열의 size는 6이면 모든 원소를 Counting할 배열을 만들수 있다.
  - 1~5, 10000을 담을 Counting 배열의 size는 10001로 만들어야하기에 비효율적

```
int arr[5] = {5, 4, 3, 2, 1};
int sorted_arr[5];

// 과정 1 - counting 배열의 사이즈를 최대값 5가 담기도록 크게 잡기
```

```
// arr idx = 0~5
int counting[6];    // 단점 : counting 배열의 사이즈의 범위를 가능힌

// 과정 2 - counting 배열의 값을 증가해주기.
// 원소별 갯수 카운팅
for (int i = 0; i < arr.length; i++) {
    counting[arr[i]]++;
}

// 과정 3 - counting 배열을 누적합으로 만들어주기.
// 누적합 생성
for (int i = 1; i < counting.length; i++) {
    counting[i] += counting[i - 1];
}
// 과정 4 - 뒤에서부터 배열을 돌면서, 해당하는 값의 인덱스에 값을 넣어주기
for (int i = arr.length - 1; i >= 0; i--) {
    counting[arr[i]]--;
    sorted_arr[counting[arr[i]]] = arr[i];
}
```