Sort

이은성

CONTENT
목차

개요

▶ 정렬

어떤 데이터들이 주어졌을 때 이를 특정 순서대로 나열하는 것

▶ 정렬하는 이유

탐색을 하기 위해

정렬 되지 않은 데이터는 순차 탐색 해야 하지만, 정렬된 데이터는 이진탐색 가능

예) 43억 건의 데이터가 정렬되어 있다면 최악의 경우 32번의 탐색만으로 찾을 수 있음

개요

▶ 정렬 종류

O(n^2)

Bubble Selection

Insertion

O(nlogn)

Merge Heap

Quick Tree

Tim Intro

Block merge

그 외

Radix Counting

shell Gravity

Sleep Bogo

BogoBogo

개요

▶ 정렬 종류

O(n²)

Bubble Selection

Insertion

O(nlogn)

Merge Heap

Quick Tree

Tim Intro

Block merge

그 외

Radix Counting

shell Gravity

Sleep Bogo

BogoBogo

개요

▶ 정렬 종류

Tim

- Merge + Insertion, 파이썬 sort 알고리즘

Intro

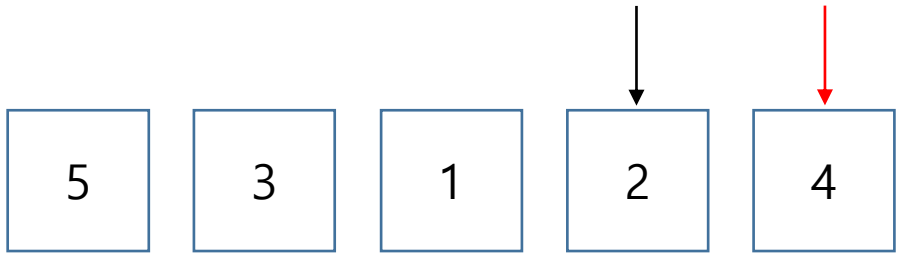
- Quick + Heap, C++ std::sort 알고리즘 (algorithm 헤더파일에 존재)

Block merge

- Merge + Insertion

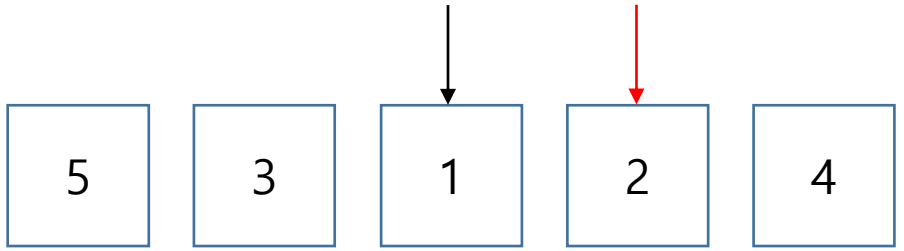
O(n²)

- Bubble Sort



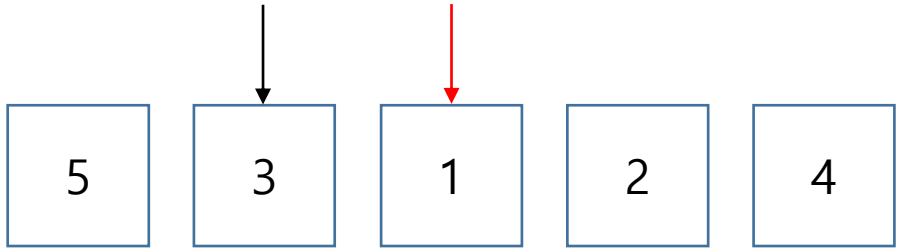
O(n²)

- Bubble Sort



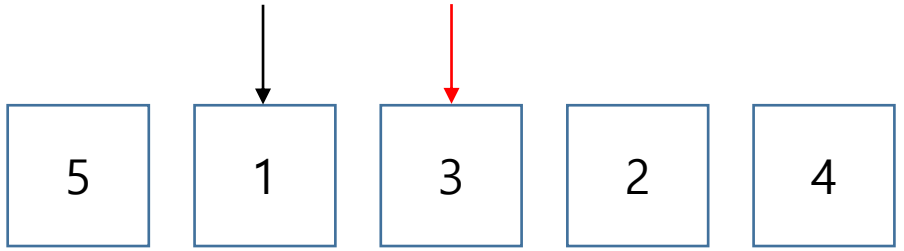
O(n²)

- Bubble Sort



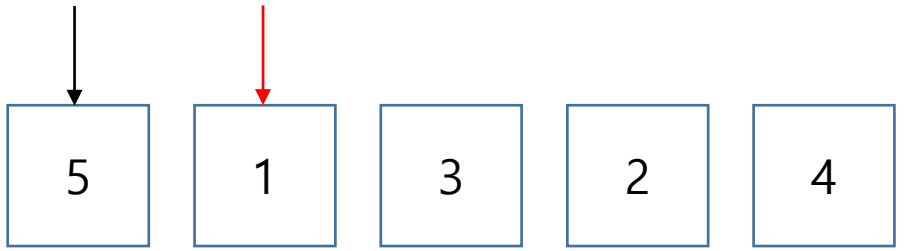
O(n²)

- Bubble Sort



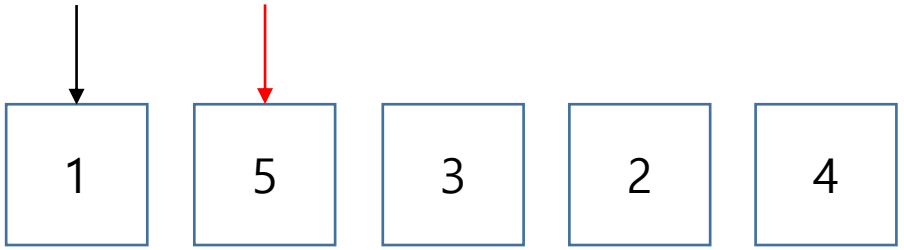
O(n²)

- Bubble Sort



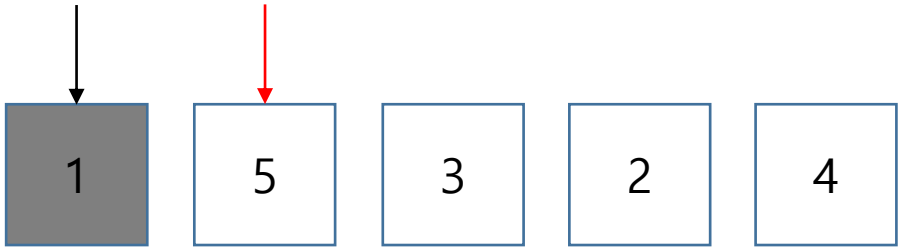
O(n²)

- Bubble Sort



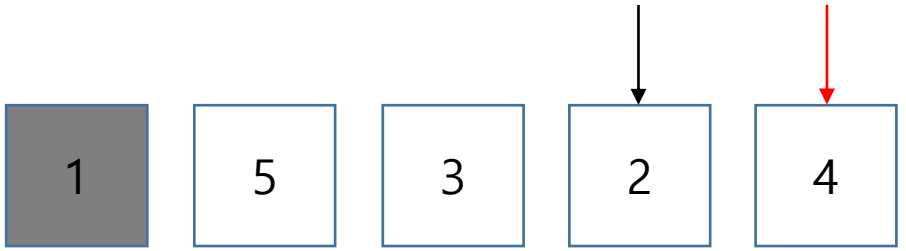
O(n²)

- Bubble Sort



O(n²)

- Bubble Sort



이후 모든 자리가 정렬될 때까지 반복하게 됨

O(n²)

- Bubble Sort

```
void mySwap(int* a, int* b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;

}
void mySort(int* arr, int start, int end) {

    for(int i = 0; i < end; i++){
        for(int j = end - 1; j > i; j--){
            if(arr[j- 1] > arr[j])
                mySwap(&arr[j], &arr[j - 1]);
        }
    }
}
```

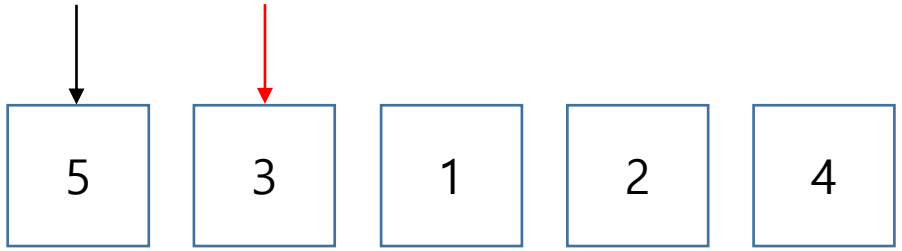
O(n²)

- Bubble Sort

  각 자리의 수는 양 옆의 수하고만 비교, 교환하게 됨

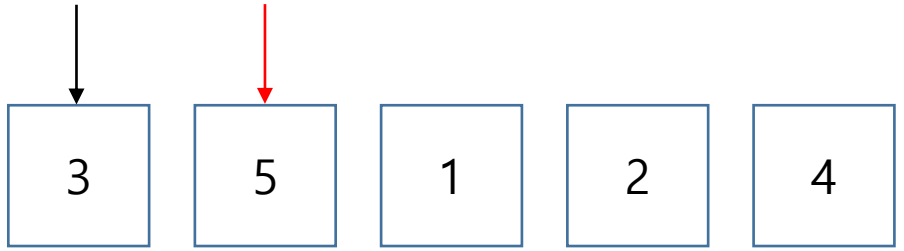
O(n²)

- Selection sort



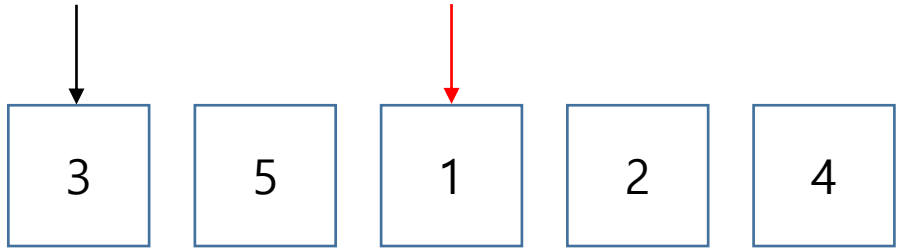
O(n²)

- Selection sort



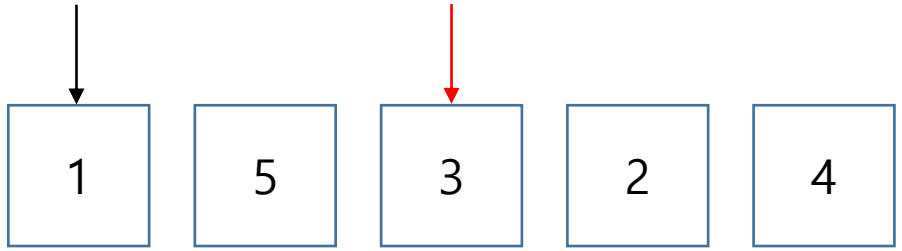
O(n²)

- Selection sort



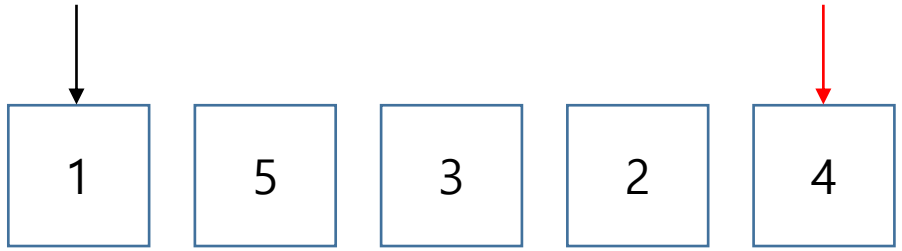
O(n²)

- Selection sort



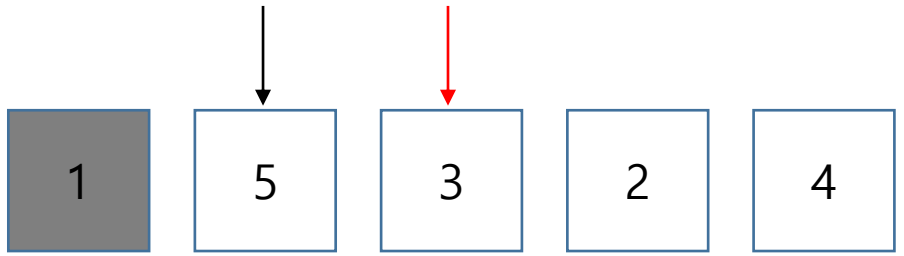
O(n²)

- Selection sort



O(n²)

- Selection sort



이후 모든 자리가 정렬될 때까지 반복하게 됨

O(n²)

- Selection sort

```
void mySwap(int* a, int* b) {
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;

}
void mySort(int* arr, int start, int end) {

    for(int i = 0; i < end; i++){
        for(int j = i + 1; j < end; j++){
            if(arr[i] > arr[j])
                mySwap(&arr[i], &arr[j]);
        }
    }
}
```

O(n²)

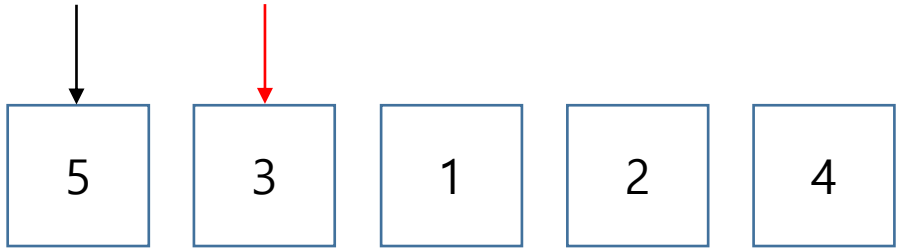
- Selection sort

  각 자리의 수는 배열 안의 모든 수와 비교하게 됨

  자리를 바꾸는 두 수 외엔 변동 없음

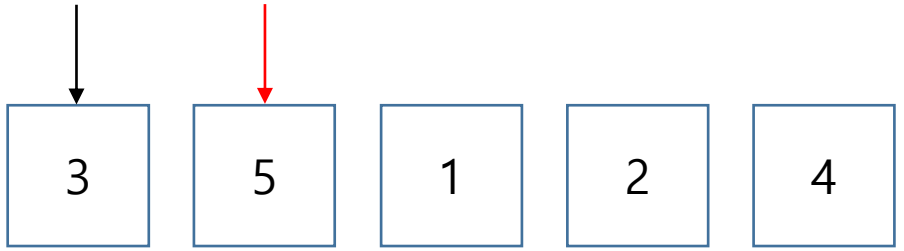
O(n²)

- Insertion sort



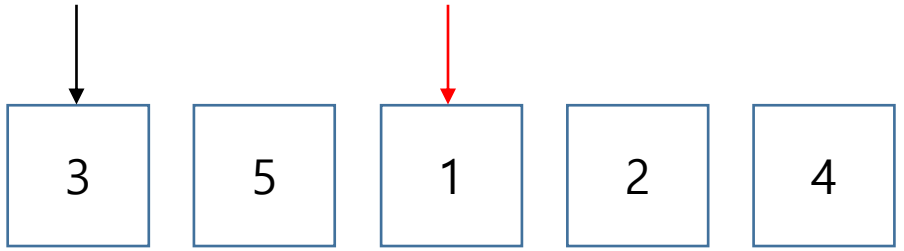
O(n²)

- Insertion sort



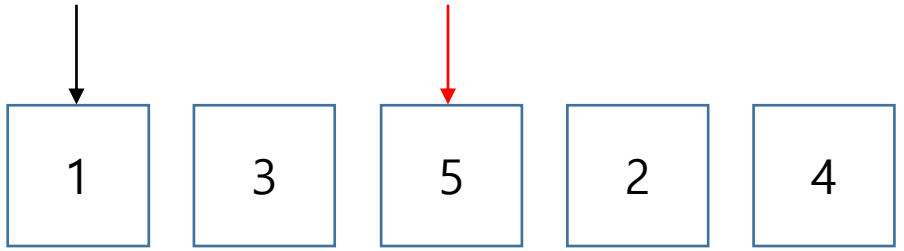
O(n²)

- Insertion sort



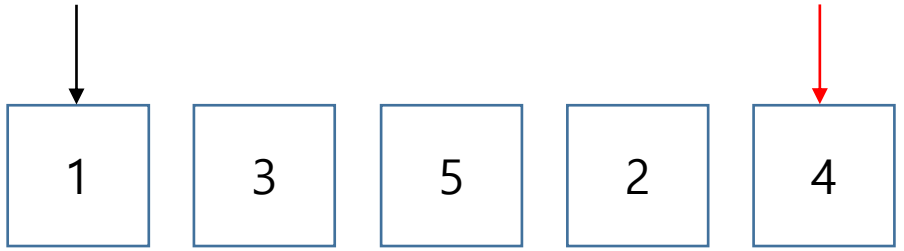
O(n²)

- Insertion sort



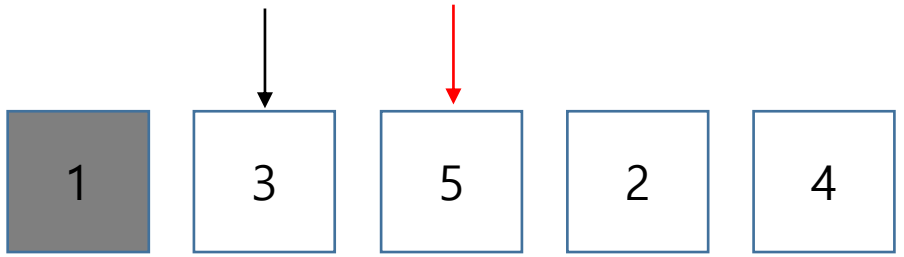
O(n²)

- Insertion sort



O(n²)

- Insertion sort



이후 모든 자리가 정렬될 때까지 반복하게 됨

O(n²)

- Insertion sort

```
void mySort(int* arr, int start, int end) {

    int tmp;

    for(int i = 0; i < end; i++){
        for(int j = i + 1; j < end; j++){
            if(arr[i] > arr[j]){
                tmp = arr[j];
                for(int k = j; k > i; k--){
                    arr[k] = arr[k - 1];
                }
                arr[i] = tmp;
            }
        }
    }
}
```

Index를 하나씩 당겨주는 부분 존재

O(n²)

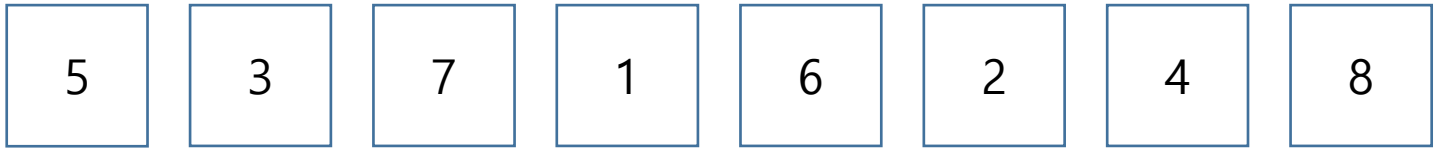
- Insertion sort

  각 자리의 수는 배열 안의 모든 수와 비교하게 됨

  자리를 바꾸게 되는 수 뒤의 수들은 한 칸씩 이동하게 됨

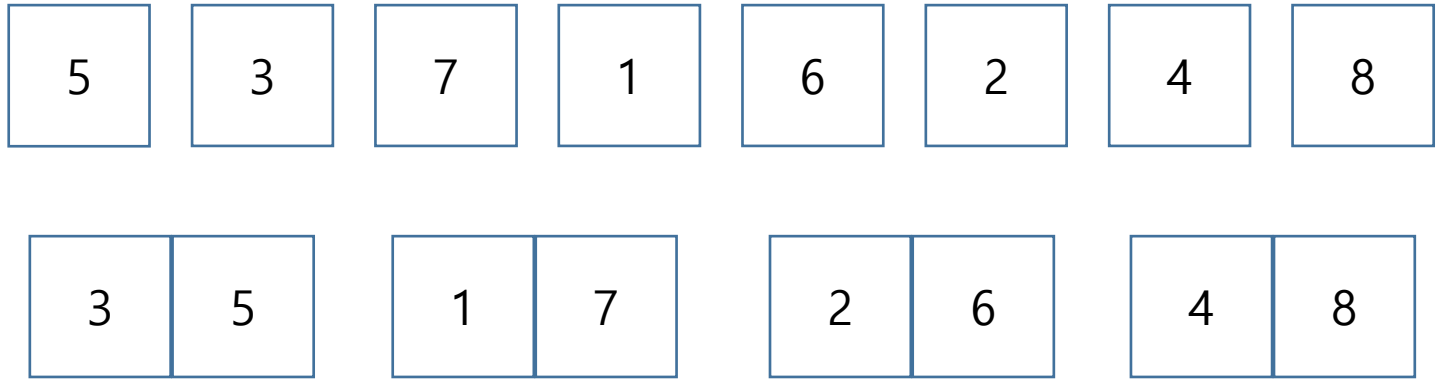
O(nlogn)

- Merge sort



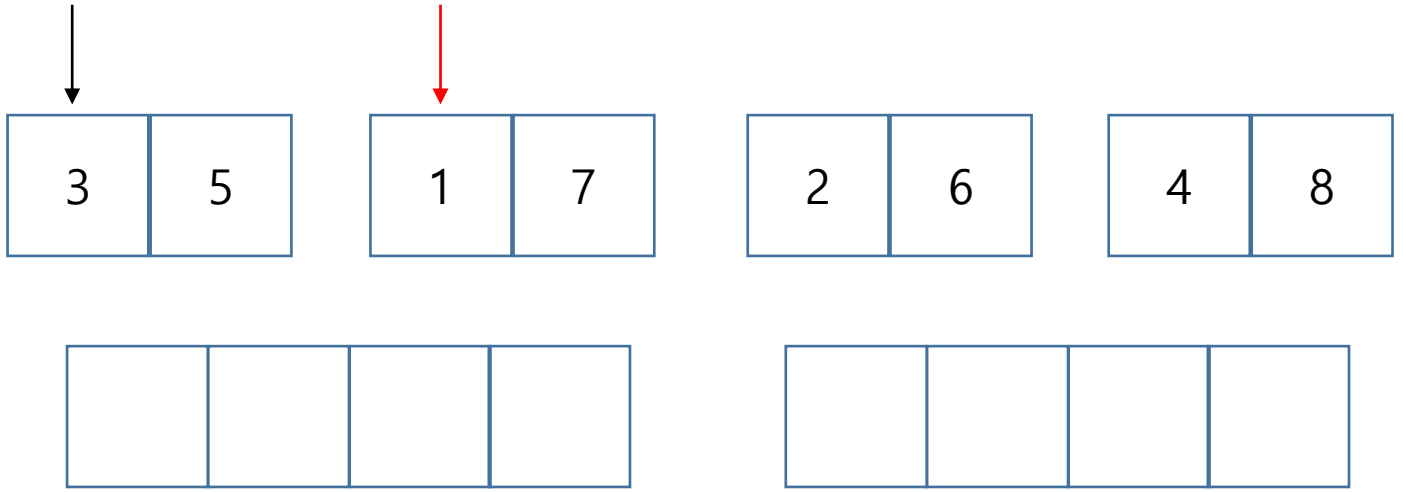
O(nlogn)

▶ Merge sort



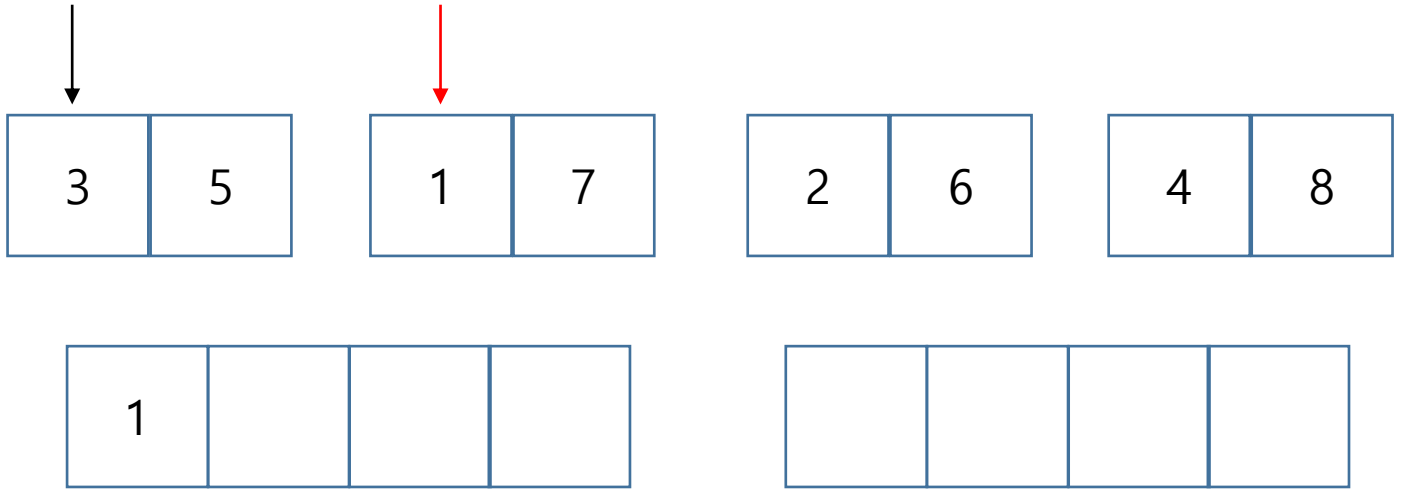
O(nlogn)

▶ Merge sort



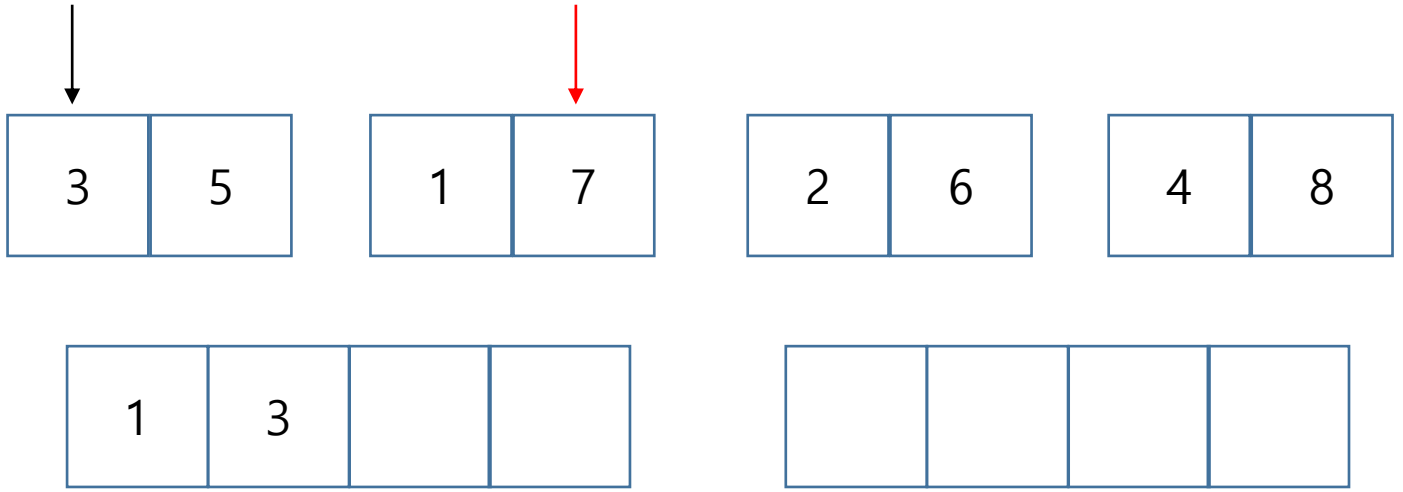
O(nlogn)

▶ Merge sort



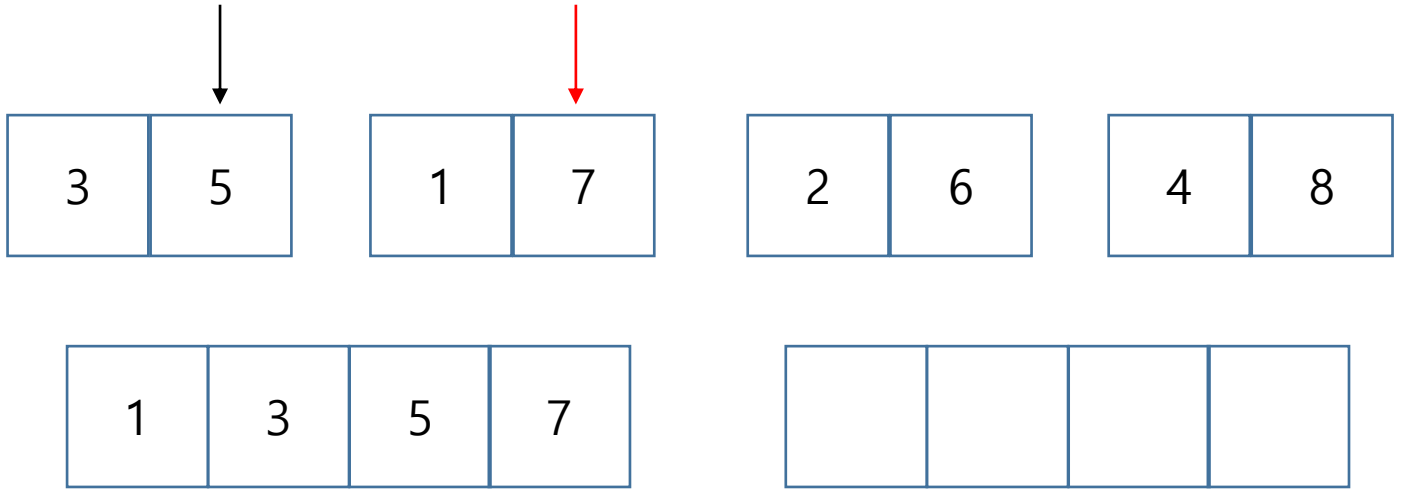
O(nlogn)

▶ Merge sort



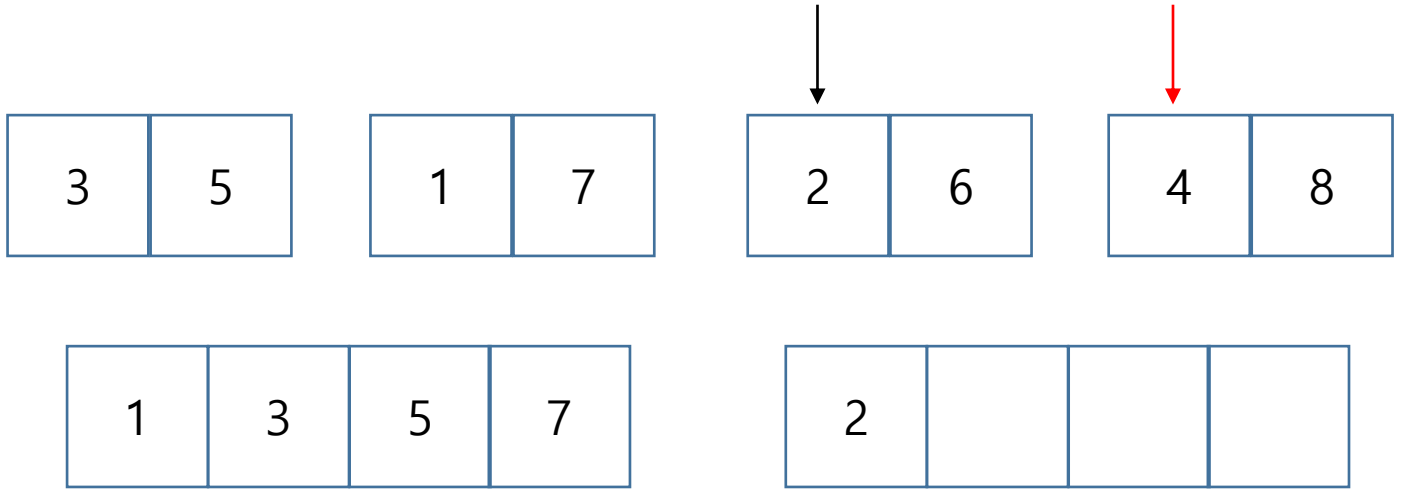
O(nlogn)

▶ Merge sort



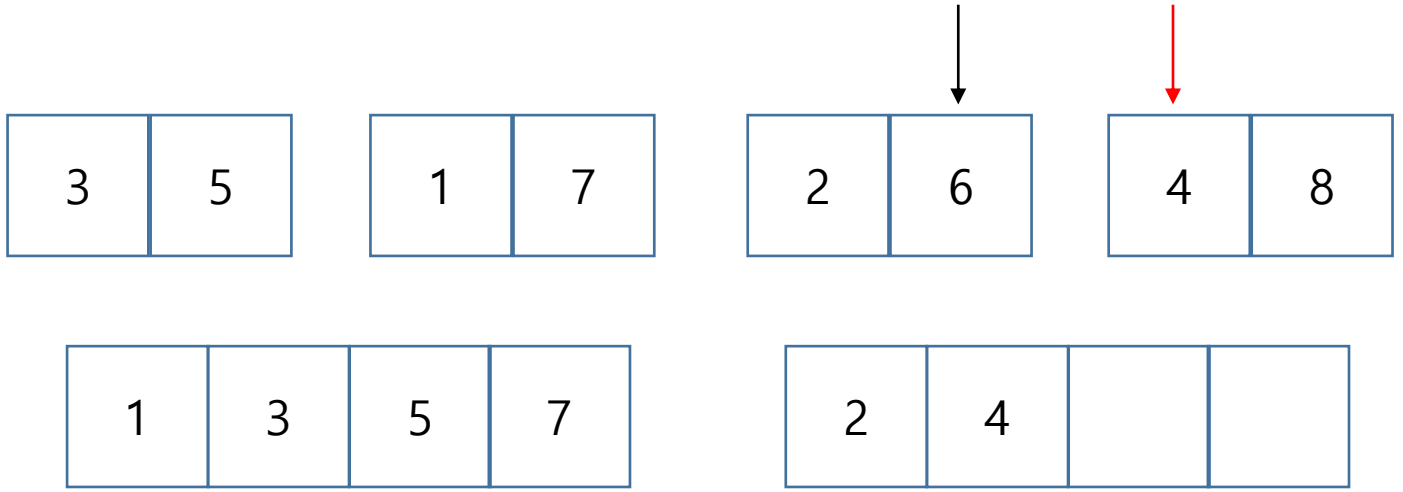
O(nlogn)

▶ Merge sort



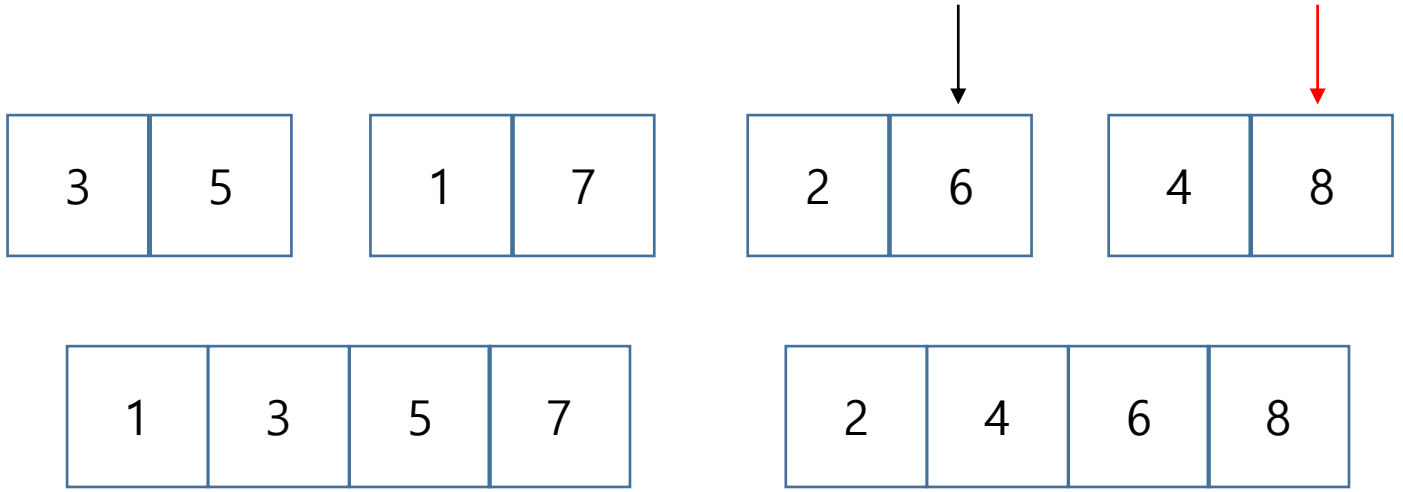
O(nlogn)

▶ Merge sort



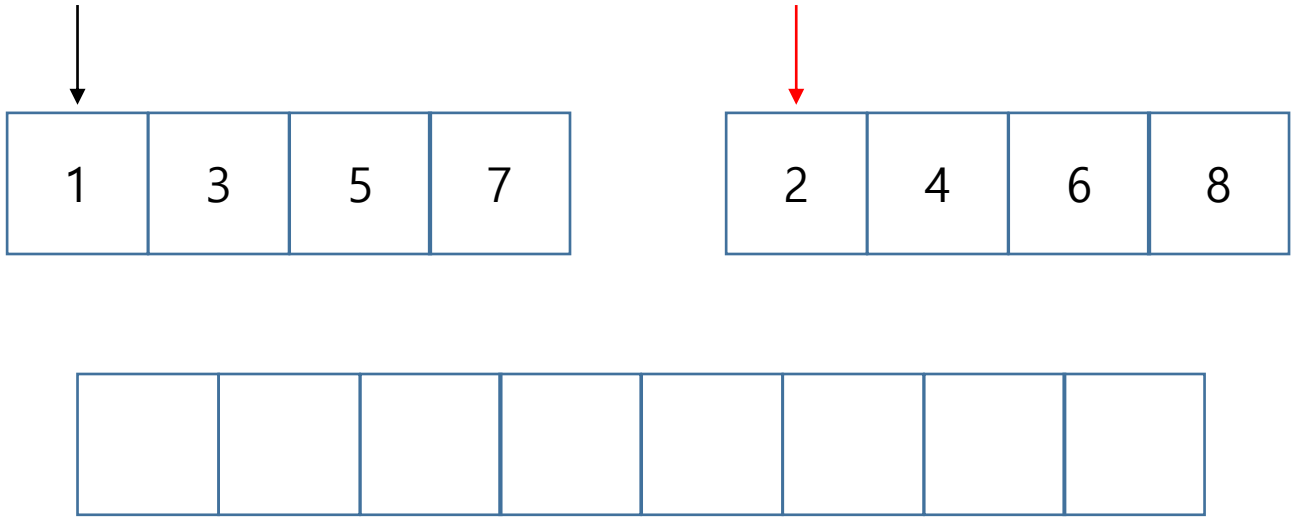
O(nlogn)

▶ Merge sort



O(nlogn)

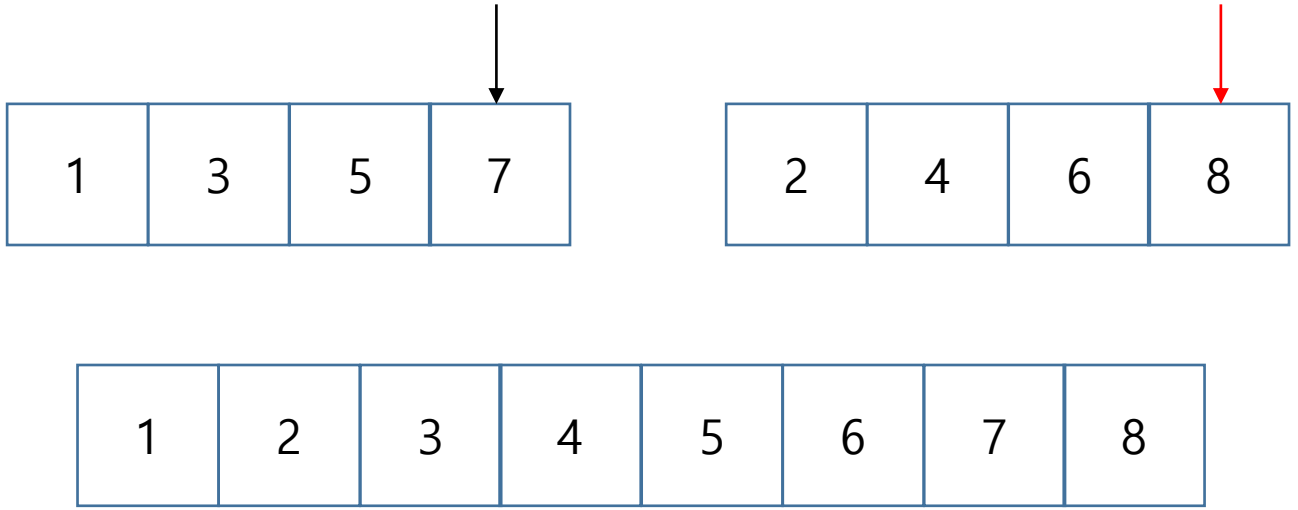
▶ Merge sort



모든 자리를 이전과 같은 순서로 채움

O(nlogn)

- Merge sort



O(nlogn)

- Merge sort

```
void mySort(int* arr, int start, int end) {
    int tmp;

    if (end - start == 1) {
        return;
    }
    else if (end - start == 2) {
        if (arr[start] > arr[end - 1]) {
            tmp = arr[start];
            arr[start] = arr[end - 1];
            arr[end - 1] = tmp;
            return;
        }
    }
    tmp = (end + start) / 2;

    mySort(arr, start, tmp);
    mySort(arr, tmp, end);
    myMerge(arr, start, tmp, end);
}
```

```
void myMerge(int* arr, int start, int mid, int end) {
    int* tmp = new int[end - start];
    int left = start, right = mid;

    for (int i = 0; i < end - start; i++) {
        if (left == mid) {
            tmp[i] = arr[right];
            right++;
        }
        else if (right == end) {
            tmp[i] = arr[left];
            left++;
        }
        else {
            if (arr[left] > arr[right]) {
                tmp[i] = arr[right];
                right++;
            }
            else {
                tmp[i] = arr[left];
                left++;
            }
        }
    }

    for (int i = 0; i < end - start; i++)
        arr[start + i] = tmp[i];

    delete tmp;
}
```

O(nlogn)

- Merge sort

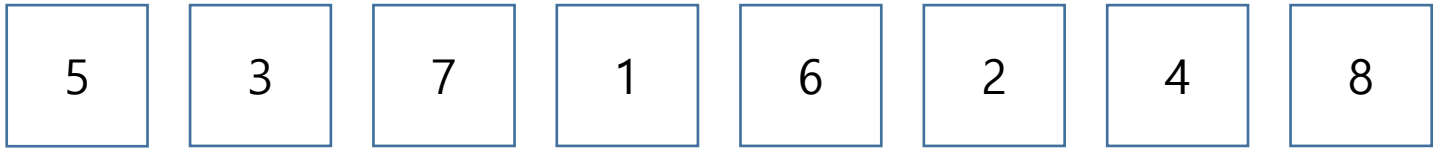
  데이터 분포에 영향 받지 않음

  최악의 경우와 최선의 경우 다 O(nlogn)으로 동일

  배열로 구현할 경우 추가메모리 필요

O(nlogn)

- Quick sort

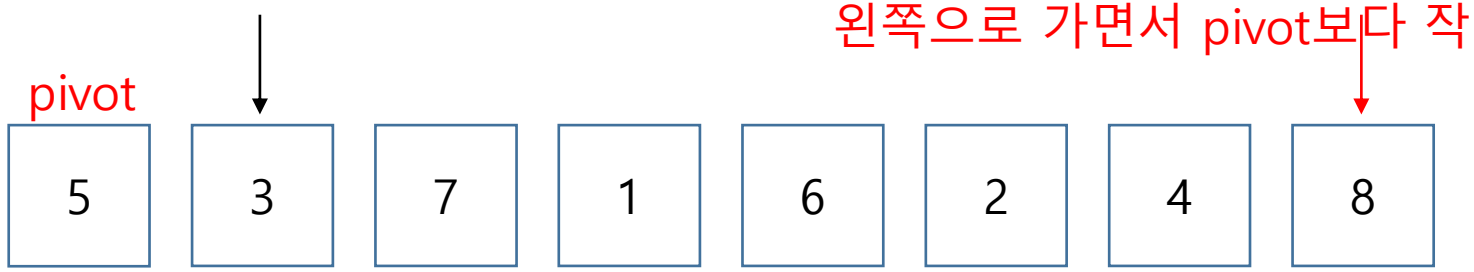


O(nlogn)

- Quick sort

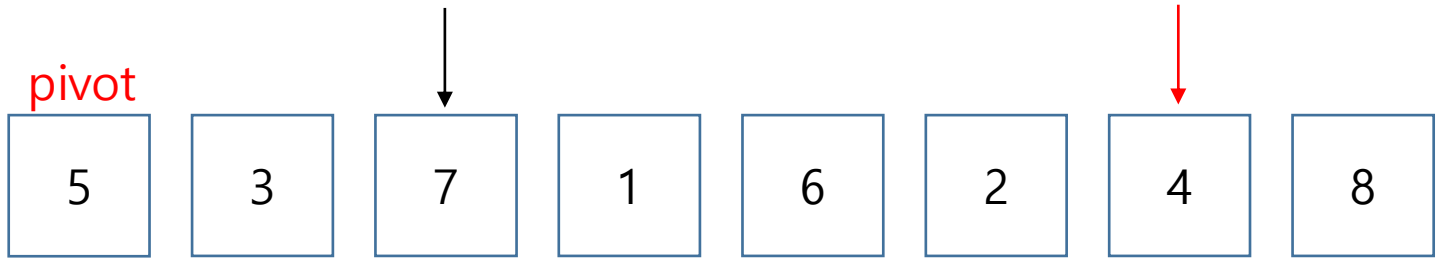
오른쪽으로 가면서 pivot보다 큰 값이 있는지 찾음

왼쪽으로 가면서 pivot보다 작은 값이 있는지 찾음



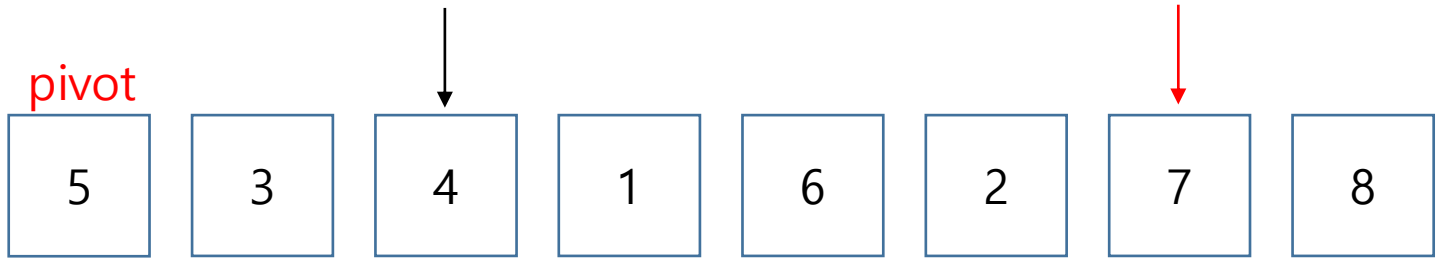
O(nlogn)

- Quick sort



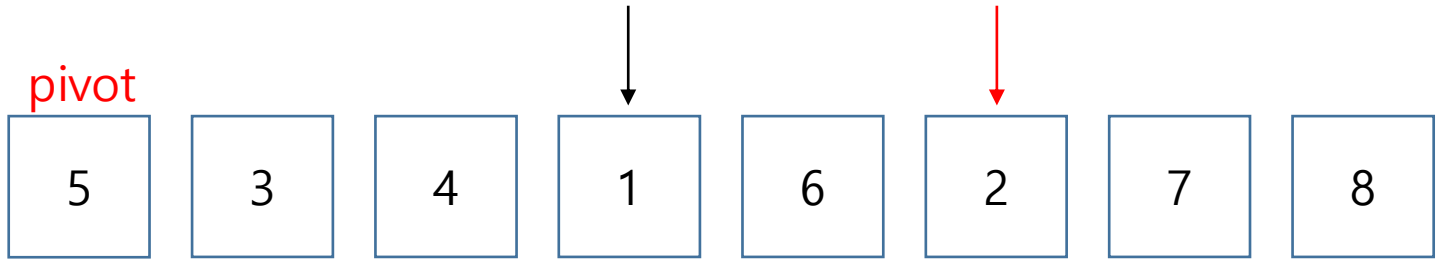
O(nlogn)

- Quick sort



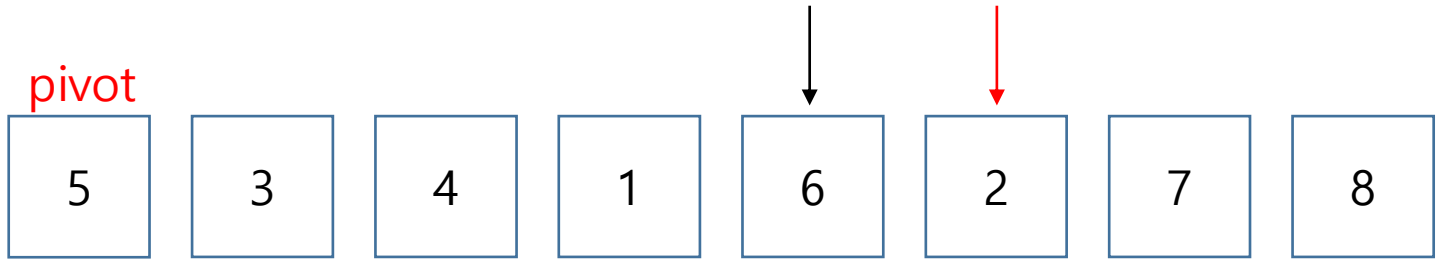
O(nlogn)

- Quick sort



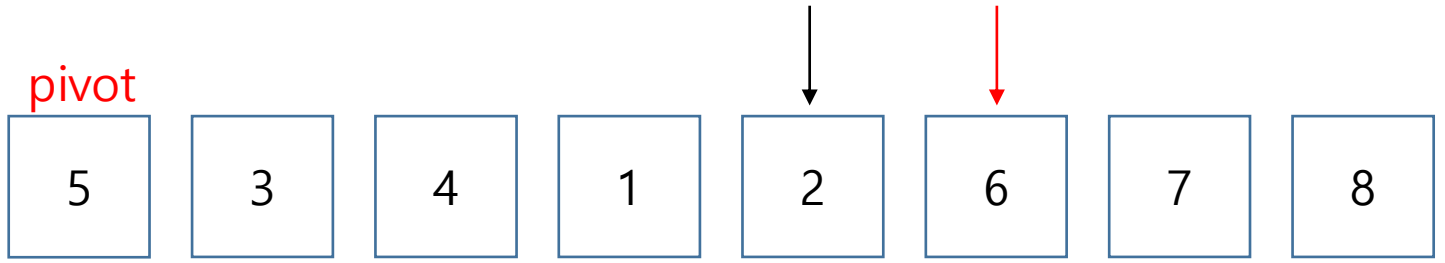
O(nlogn)

- Quick sort



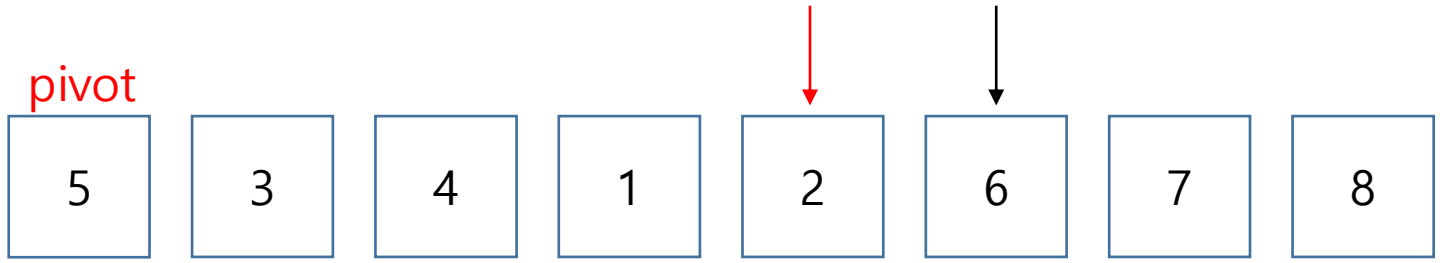
O(nlogn)

- Quick sort



O(nlogn)

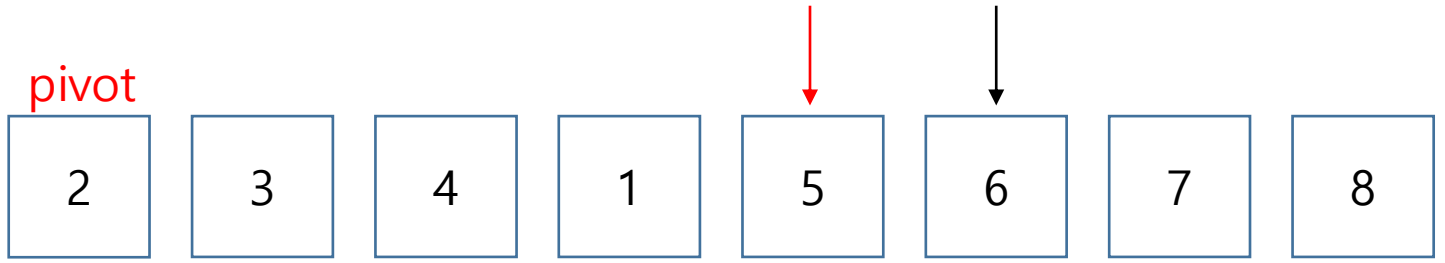
▶ Quick sort



화살표가 엇갈리면 거기서 stop하고 빨간색 화살표와 pivot 자리 바꿈

O(nlogn)

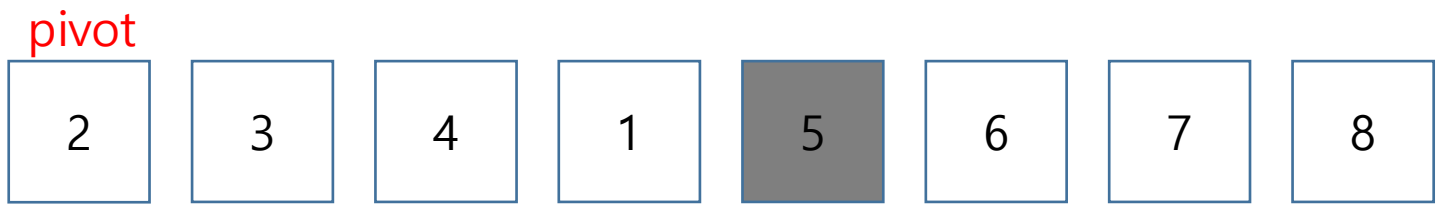
▶ Quick sort



화살표가 엇갈리면 거기서 stop하고 빨간색 화살표와 pivot 자리 바꿈

O(nlogn)

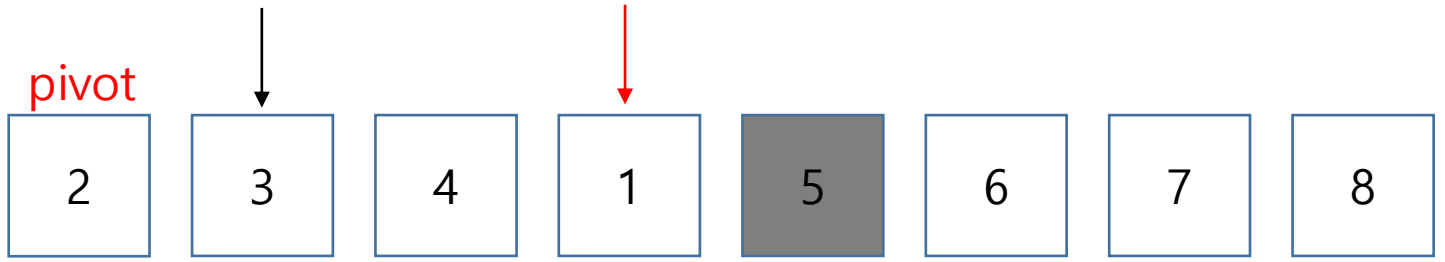
▶ Quick sort



화살표가 엇갈리면 거기서 stop하고 빨간색 화살표와 pivot 자리 바꿈

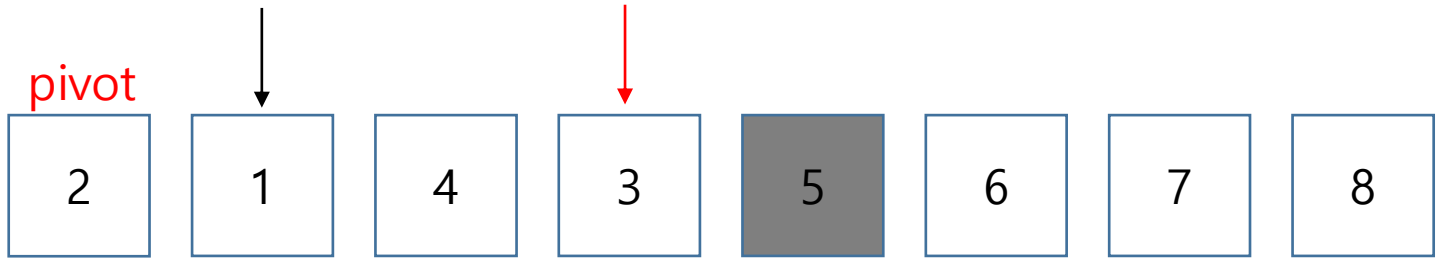
O(nlogn)

- Quick sort



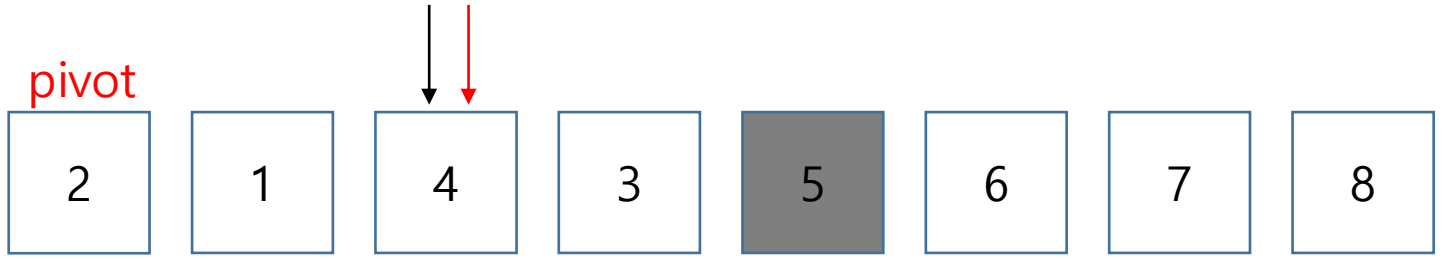
O(nlogn)

▶ Quick sort



O(nlogn)

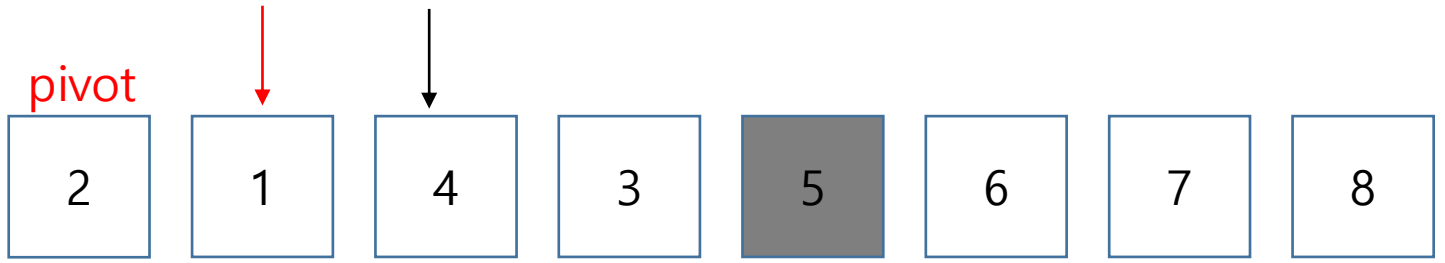
▶ Quick sort



검은색 화살표는 pivot 보다 큰 값을 찾았으니 정지
빨간색 화살표만 이동

O(nlogn)

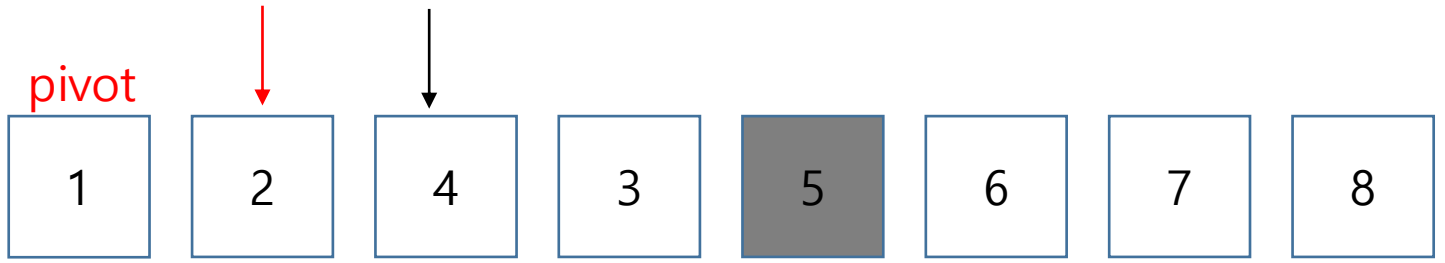
▶ Quick sort



화살표가 엇갈렸으니 빨간색 화살표와 pivot을 교환

O(nlogn)

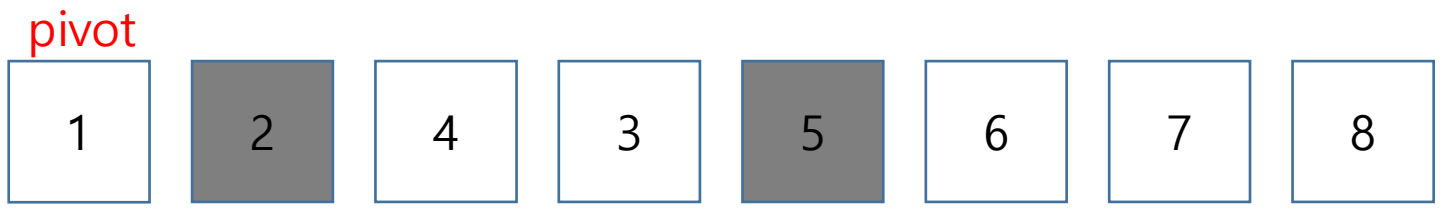
▶ Quick sort



화살표가 엇갈렸으니 빨간색 화살표와 pivot을 교환

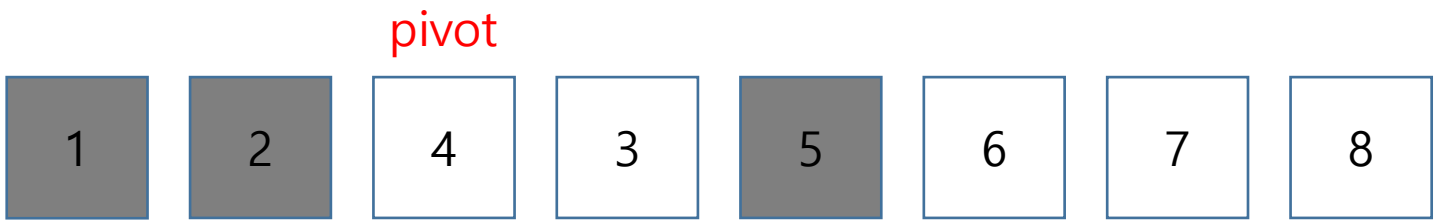
O(nlogn)

- Quick sort



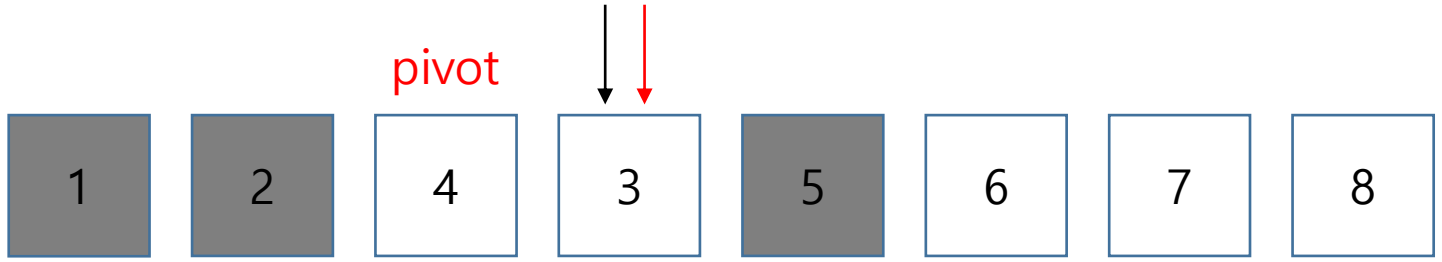
O(nlogn)

- Quick sort



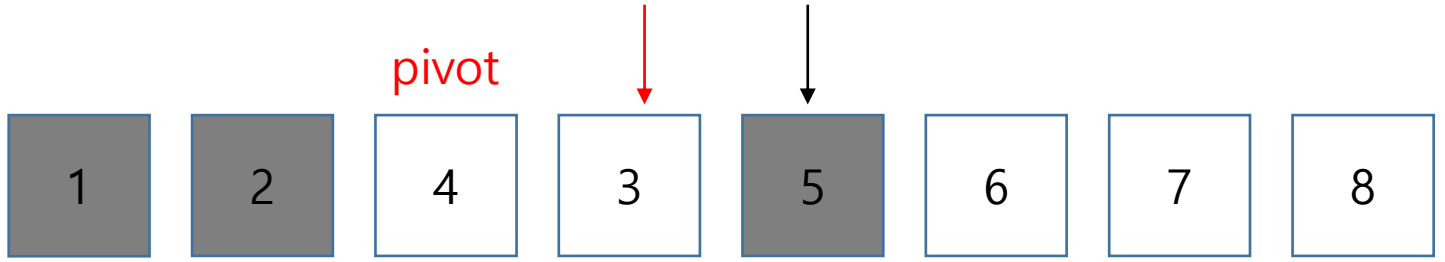
O(nlogn)

- Quick sort



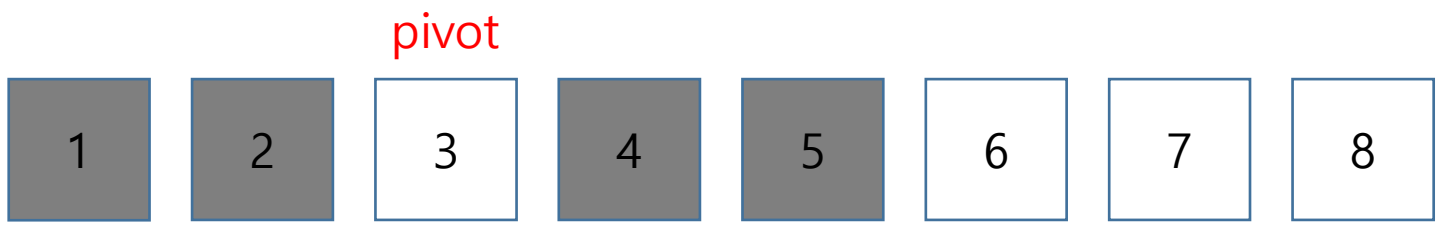
O(nlogn)

- Quick sort



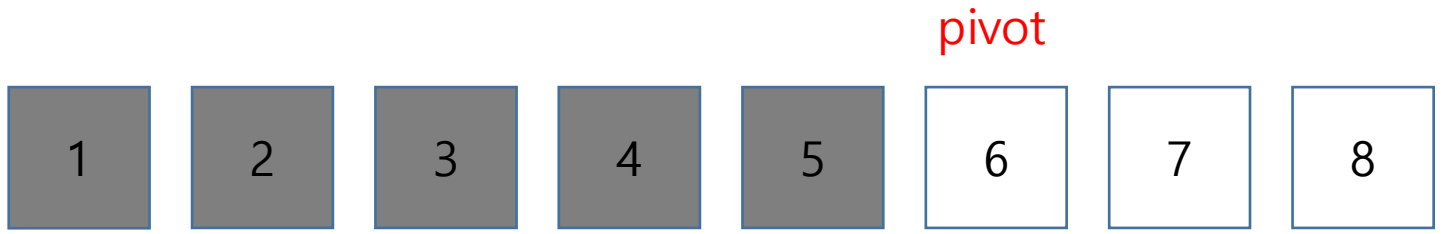
O(nlogn)

- Quick sort



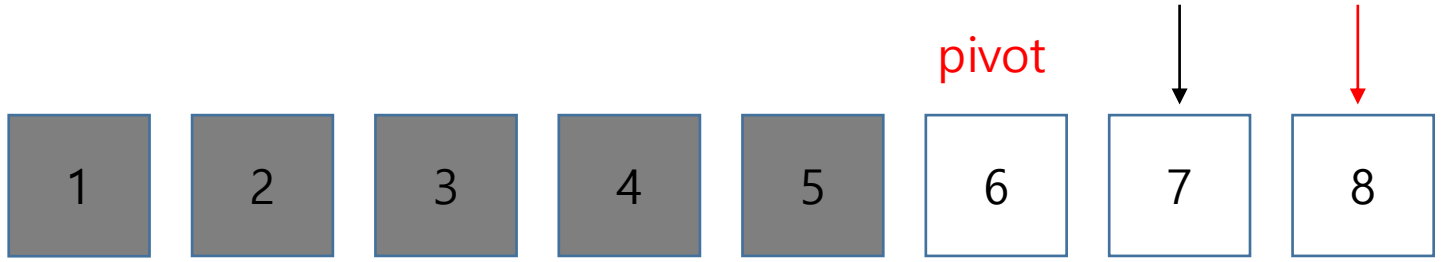
O(nlogn)

- Quick sort



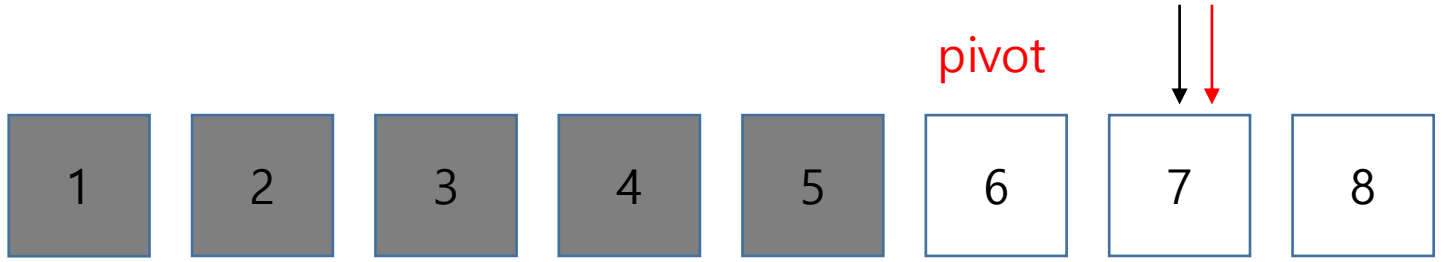
O(nlogn)

- Quick sort



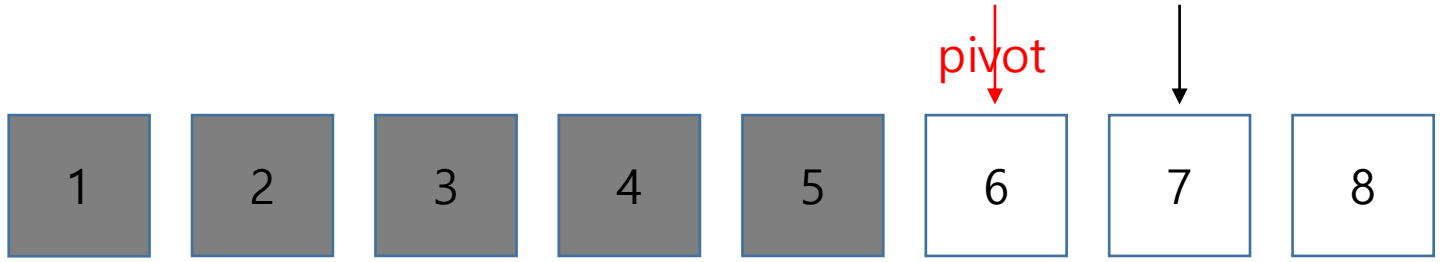
O(nlogn)

- Quick sort



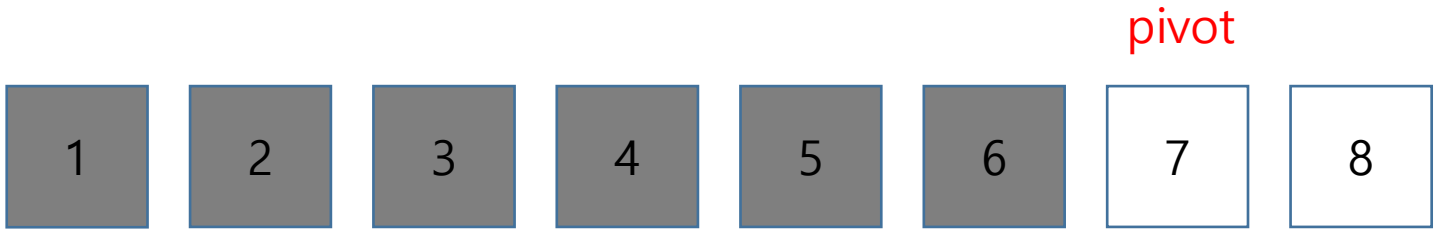
O(nlogn)

▶ Quick sort



O(nlogn)

▶ Quick sort



이후 모든 자리가 정렬될 때까지 반복하게 됨

O(nlogn)

- Quick sort

```
void mySort(int* arr, int start, int end) {

    int pivot = start;
    int i = start + 1;
    int j = end - 1;
    bool icheck, jcheck;

    if (i > j)
        return;
    while (i <= j) {
        icheck = false;
        jcheck = false;
        if (arr[i] < arr[pivot]) {
            i++;
            icheck = true;
        }
        if (arr[j] > arr[pivot]) {
            j--;
            jcheck = true;
        }
        if (!icheck && !jcheck) {
            mySwap(&arr[i], &arr[j]);
        }
    }

    mySwap(&arr[pivot], &arr[j]);
    if (j - start > 1)
        mySort(arr, start, j);
    if (end - j + 1 > 1)
        mySort(arr, j + 1, end);
}
```

O(nlogn)

- Quick sort

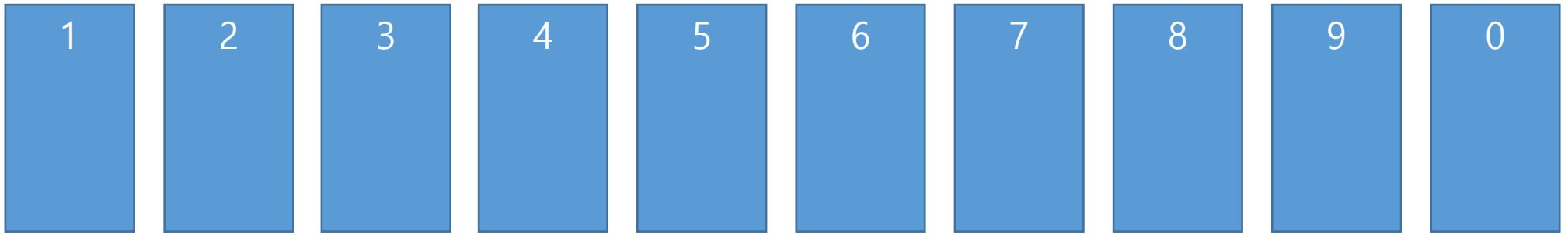
  평균적으로 다른 O(nlogn) 정렬보다 빠르다

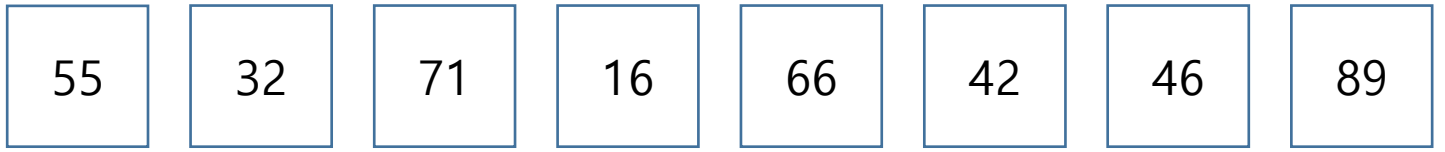
  추가로 메모리를 필요로 하지 않는다

  최악의 경우 O(n^2) 까지 걸릴 수 있다

기타

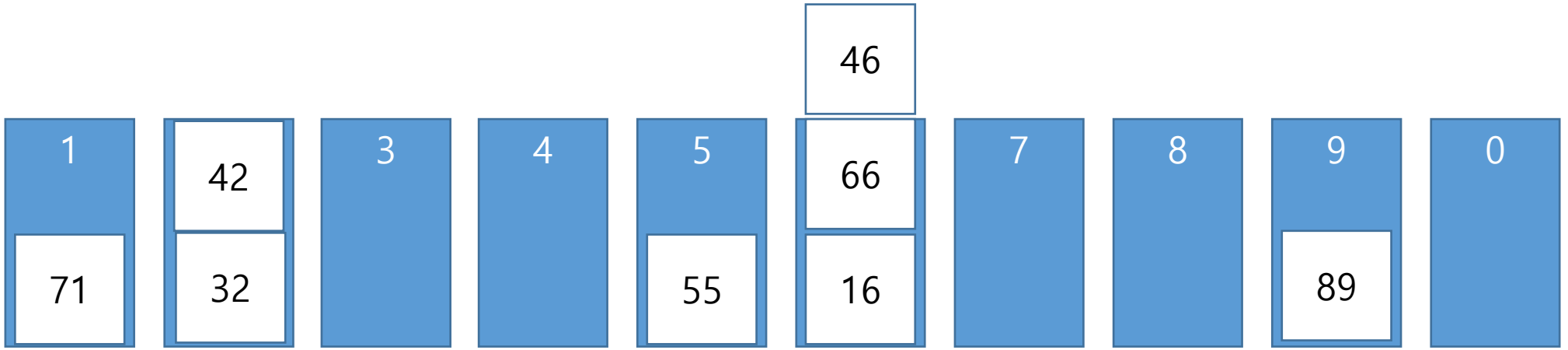
- Radix sort





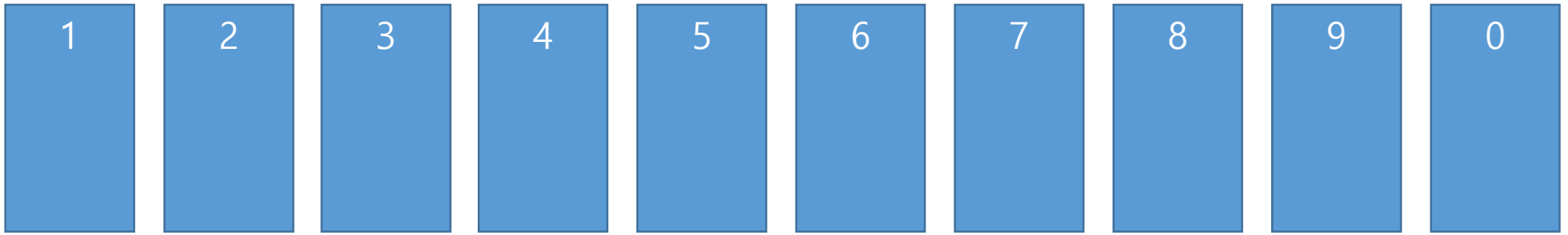
기타

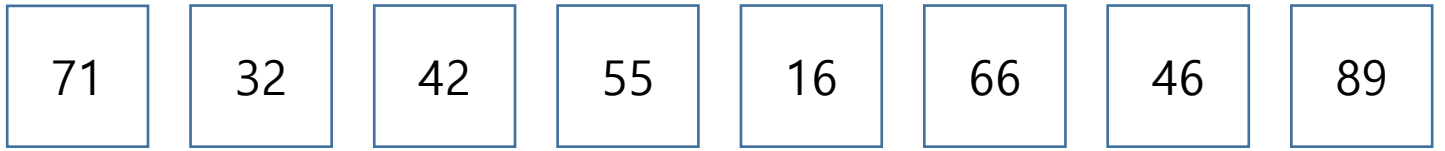
- Radix sort



기타

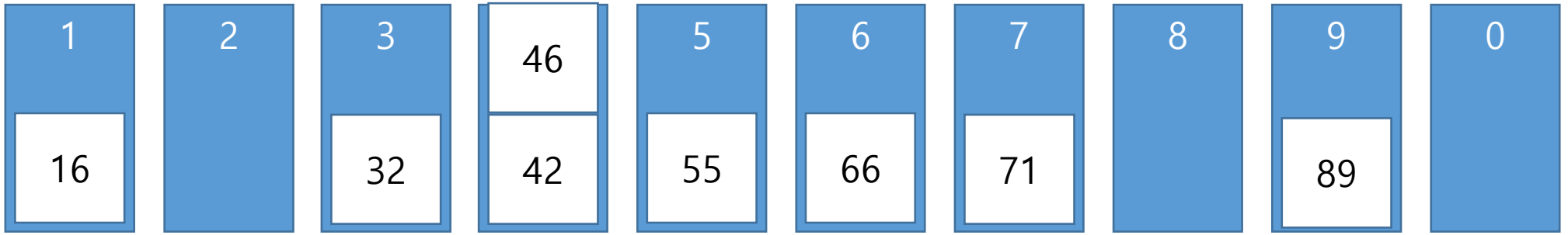
- Radix sort





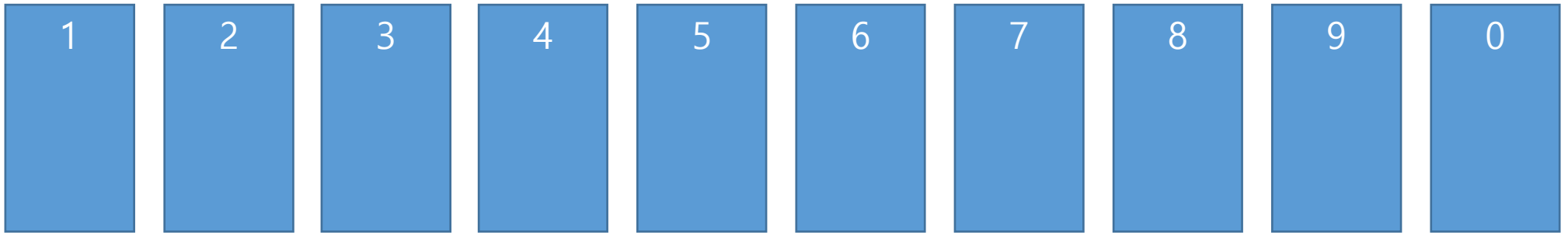
기타

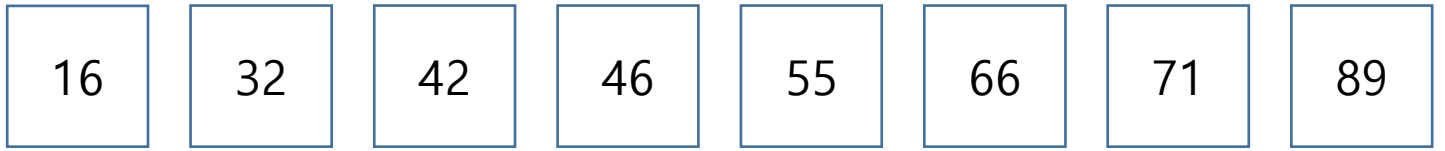
- Radix sort



기타

- Radix sort





기타

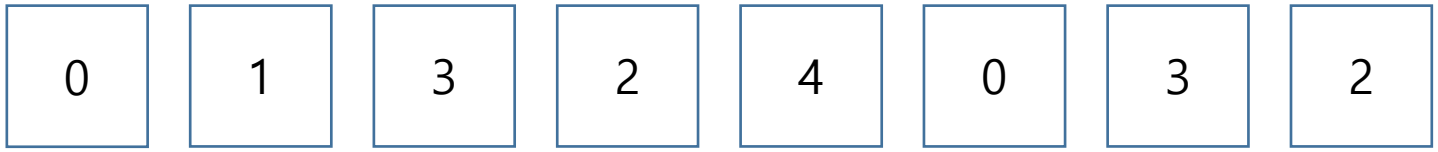
- Radix sort

  시간복잡도가 (dn)임 (d = 최대값의 자리 수)

  N이 크고 최대값이 작은 경우 nlogn보다 유리 할 수 있음

기타

- Counting sort



기타

- Counting sort

0	1	3	2	4	0	3	2

수	0	1	2	3	4

기타

▶ Counting sort

0	1	3	2	4	0	3	2

수	0	1	2	3	4
count	2	1	2	2	1

기타

▶ Counting sort

0	1	3	2	4	0	3	2

수	0	1	2	3	4
count	2	1	2	2	1
누적합	2	3	5	7	8

기타

▶ Counting sort

기타

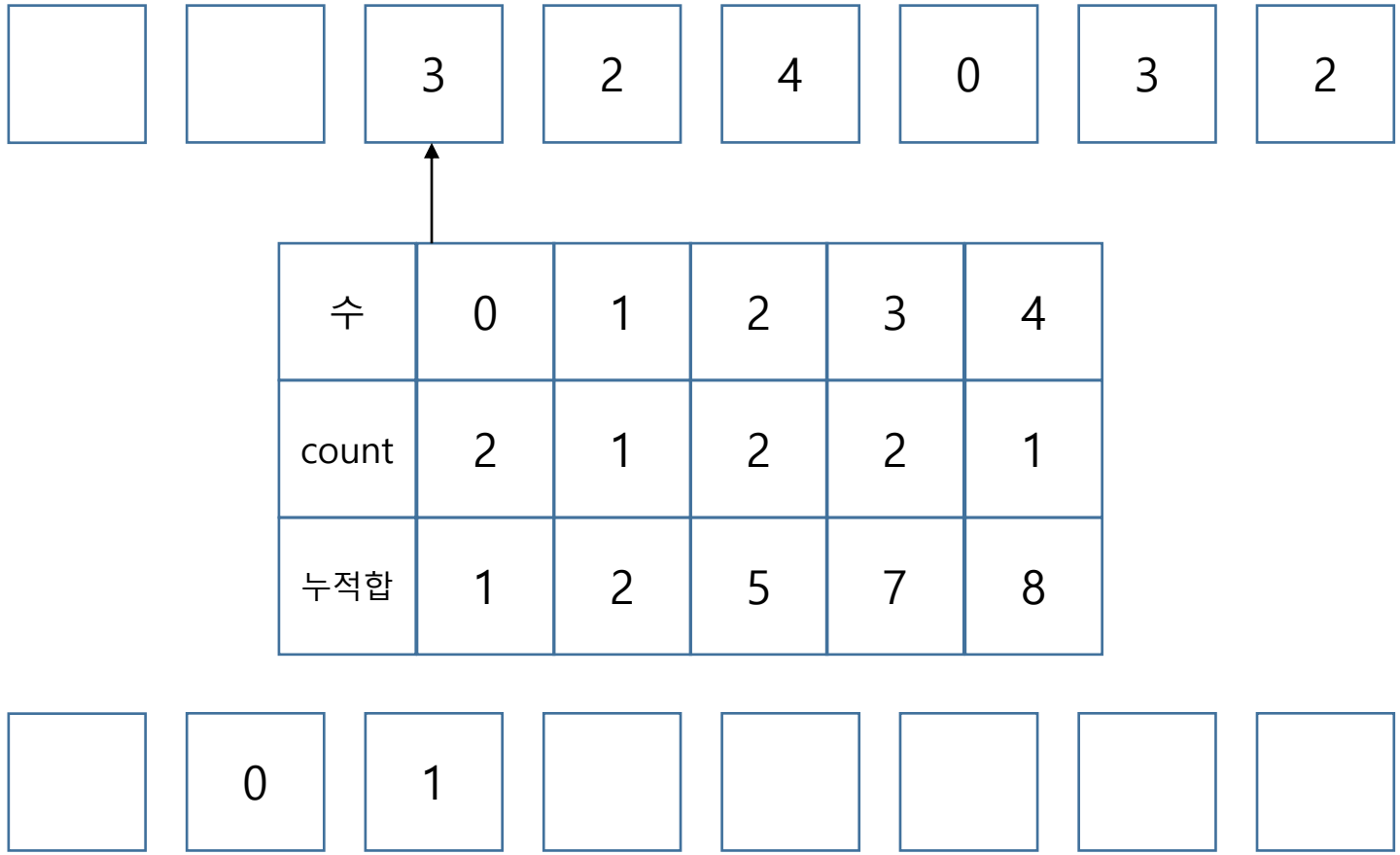
▶ Counting sort

기타

▶ Counting sort

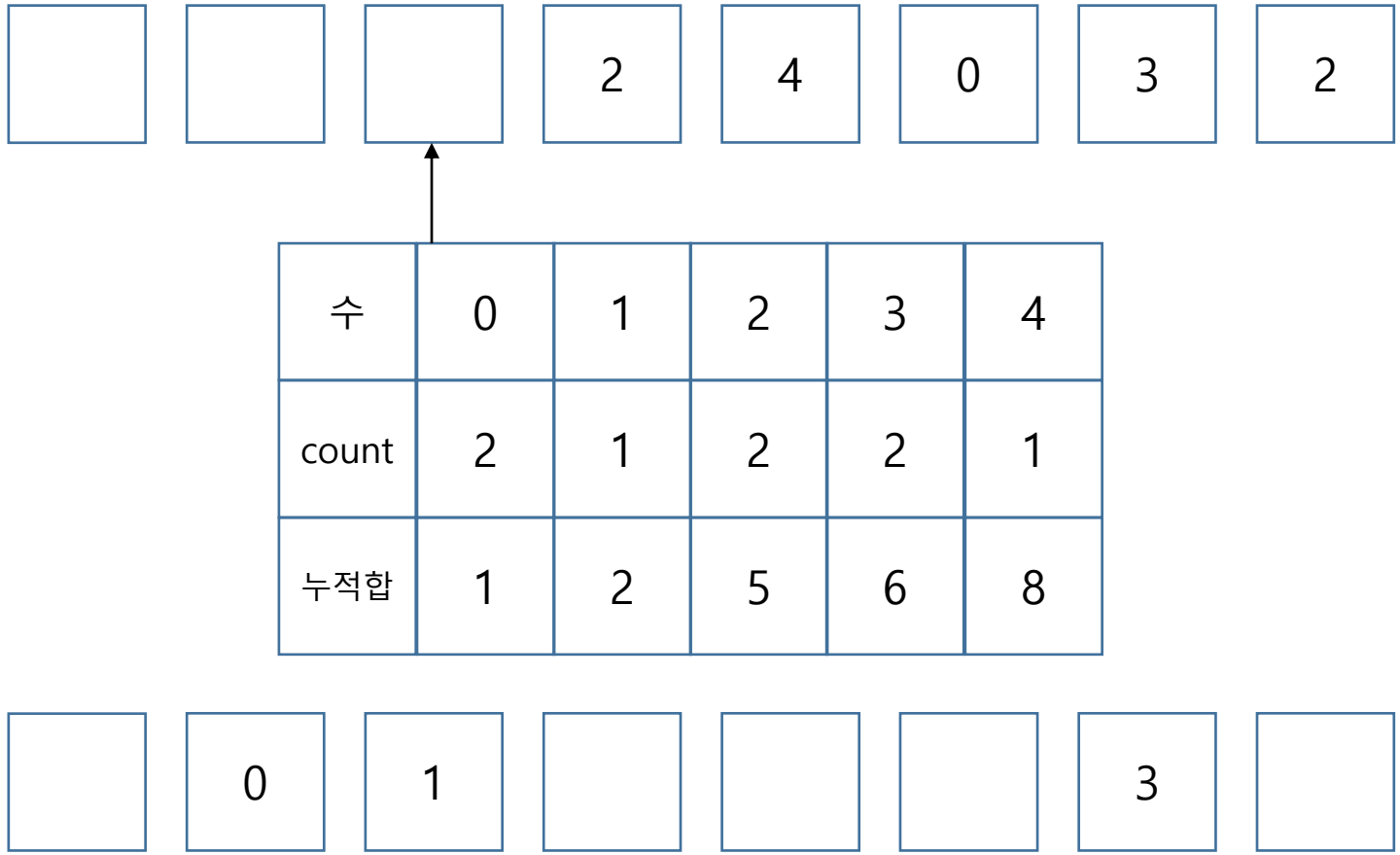
기타

▶ Counting sort



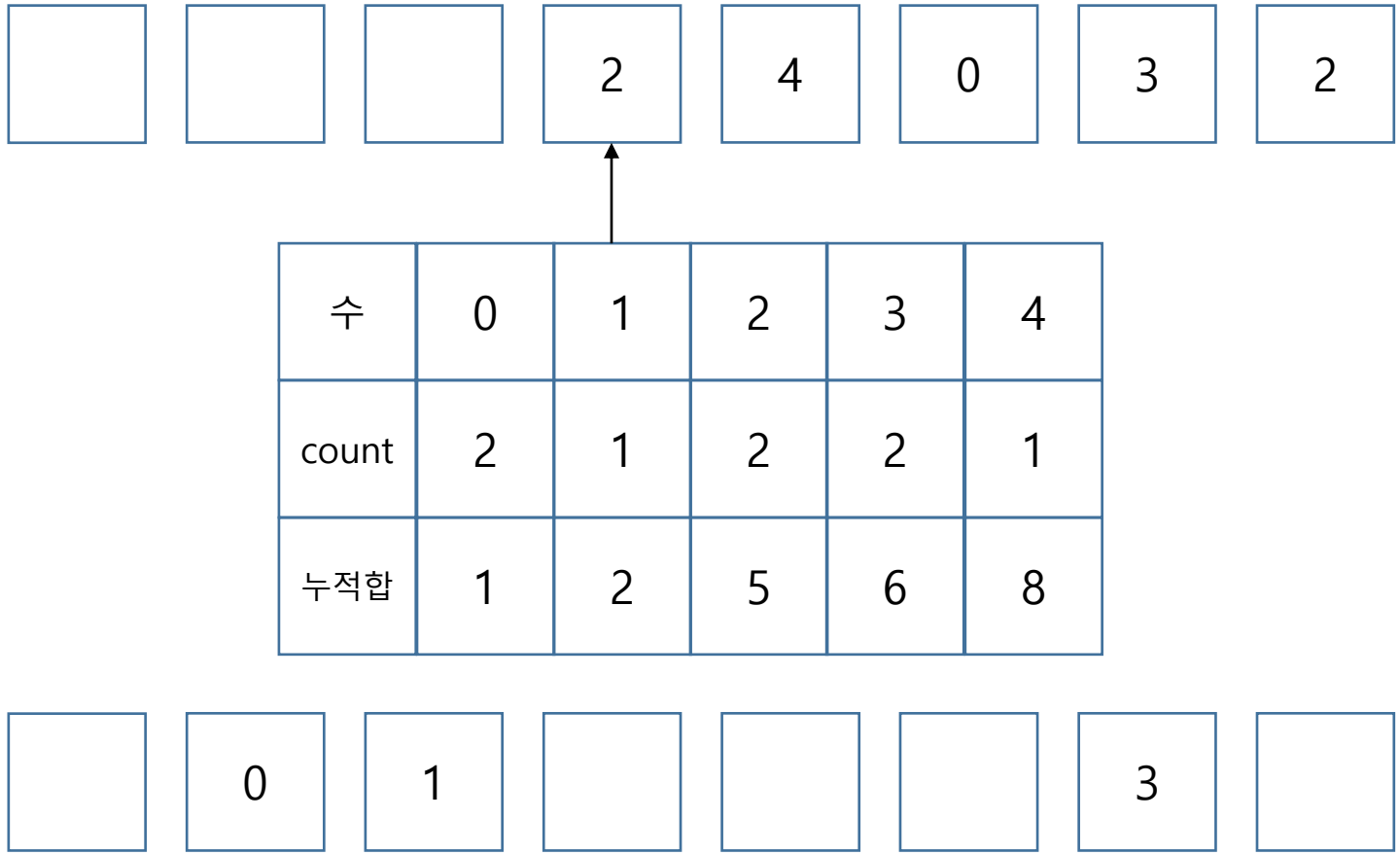
기타

▶ Counting sort



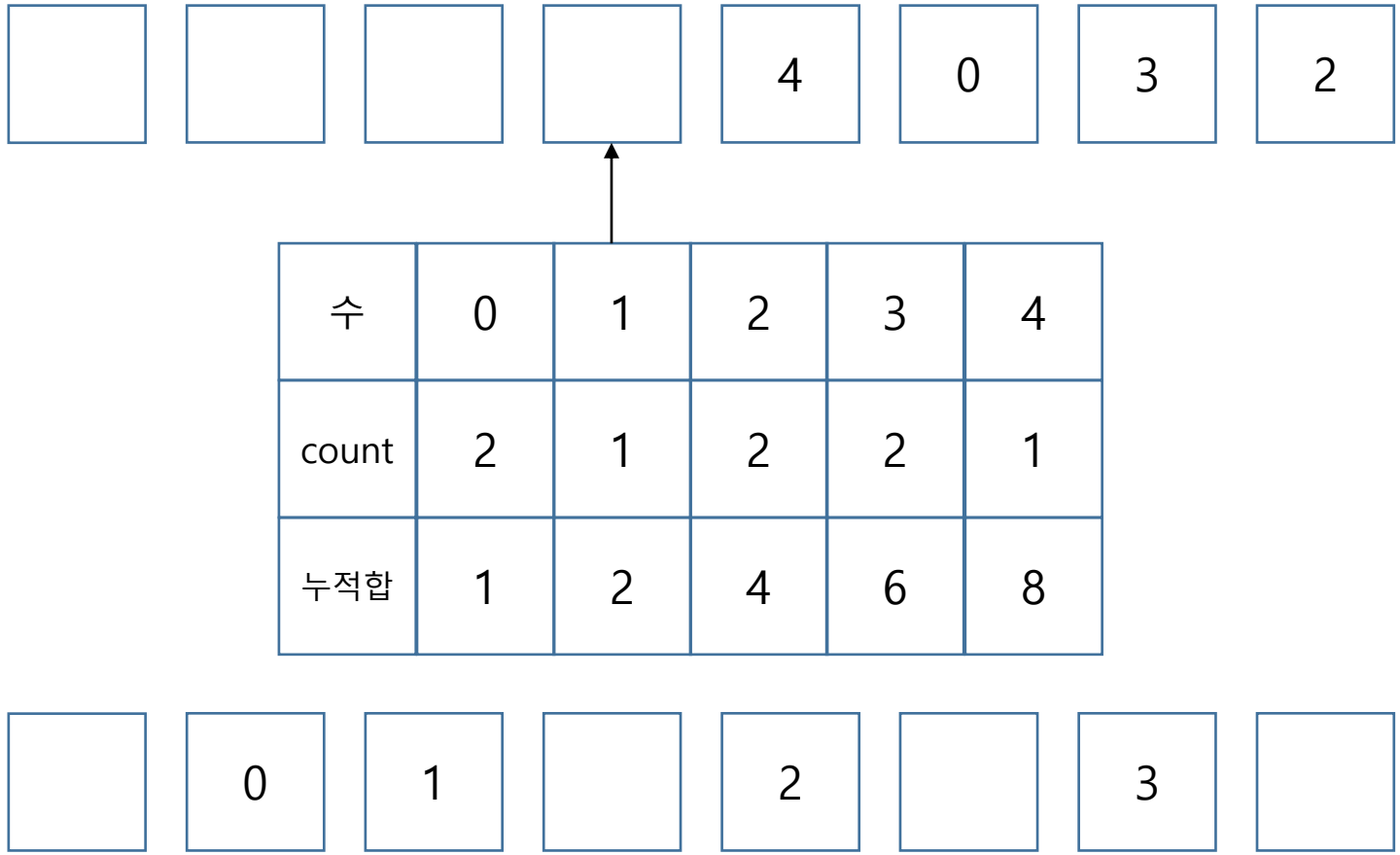
기타

▶ Counting sort



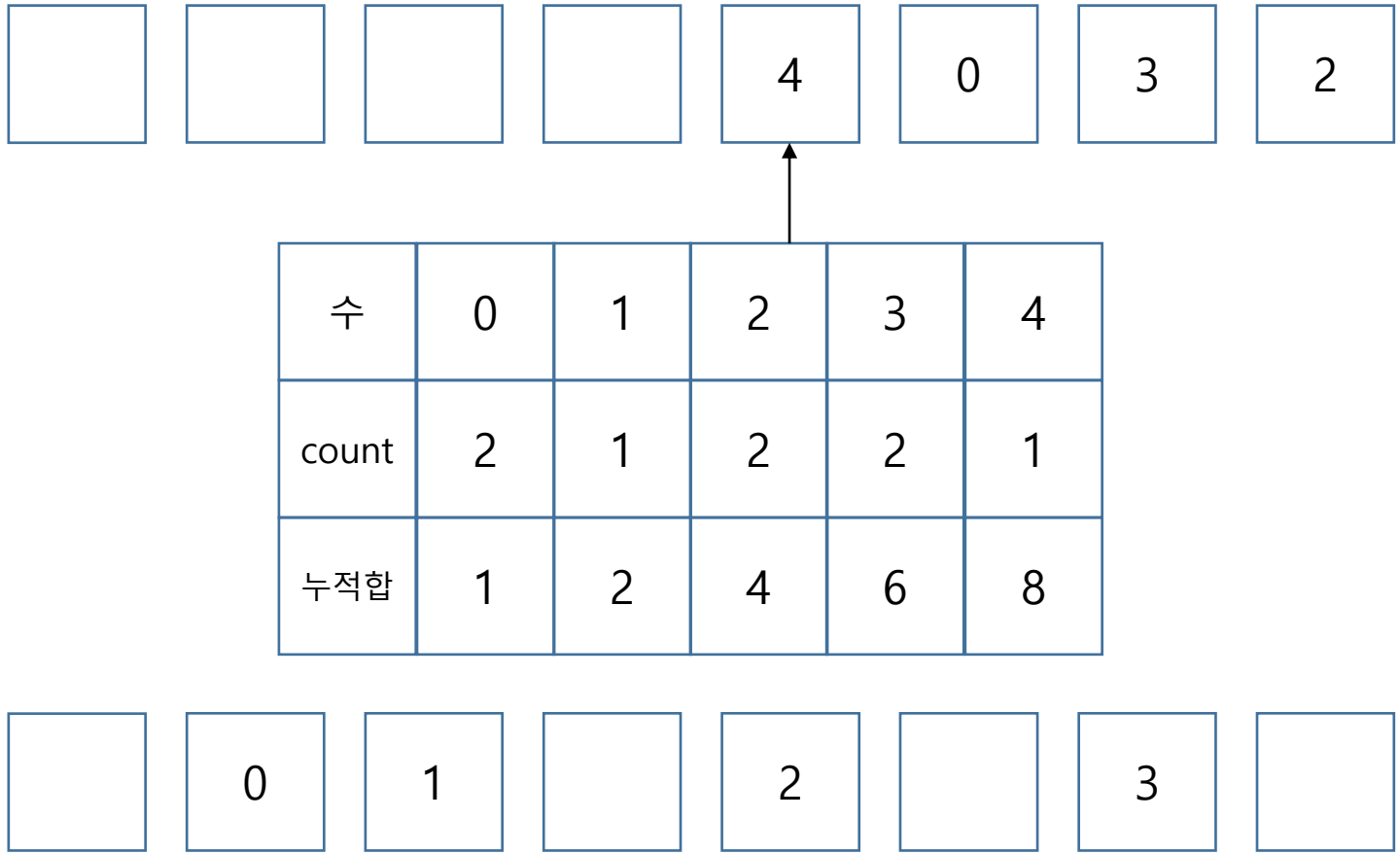
기타

▶ Counting sort



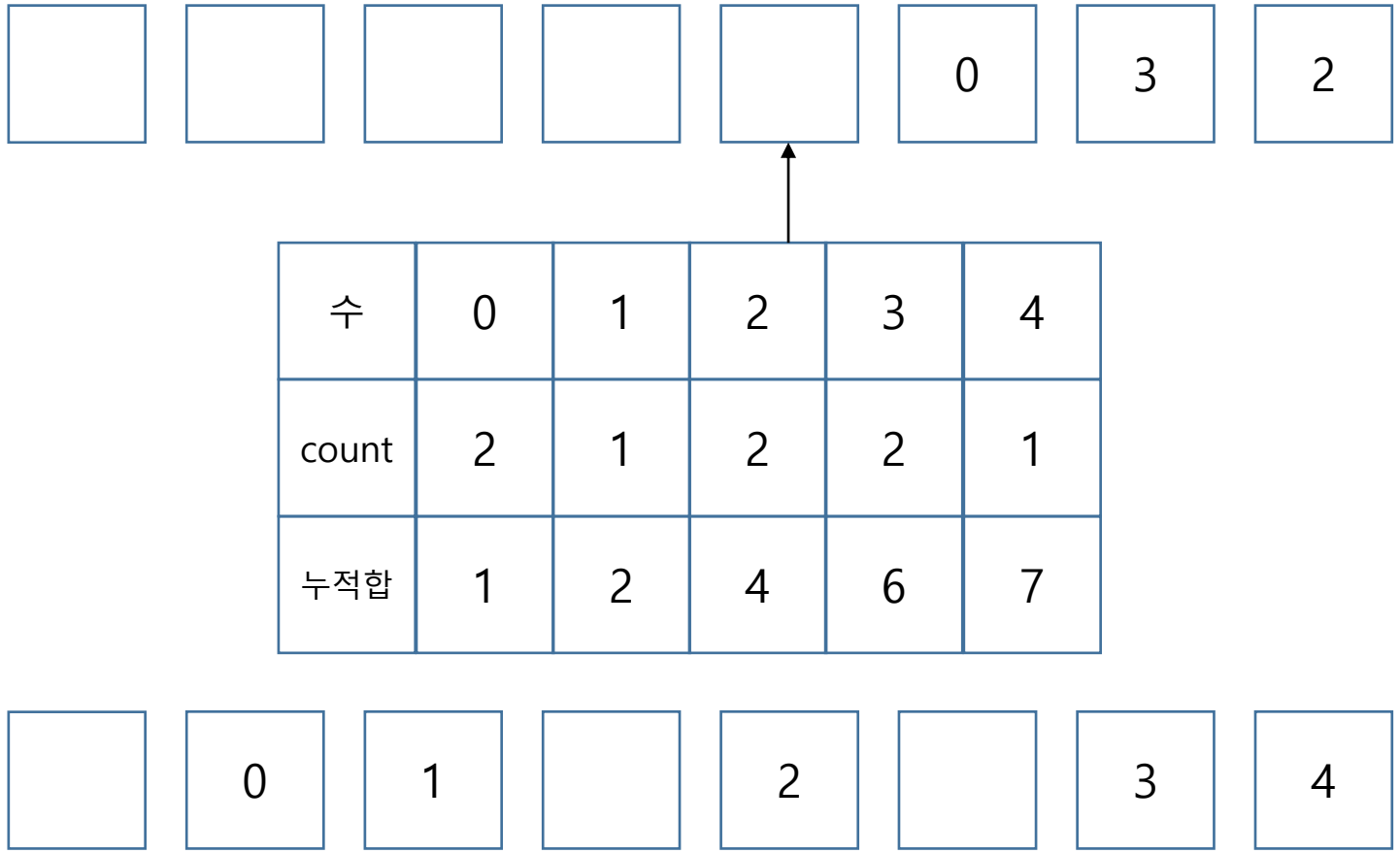
기타

▶ Counting sort



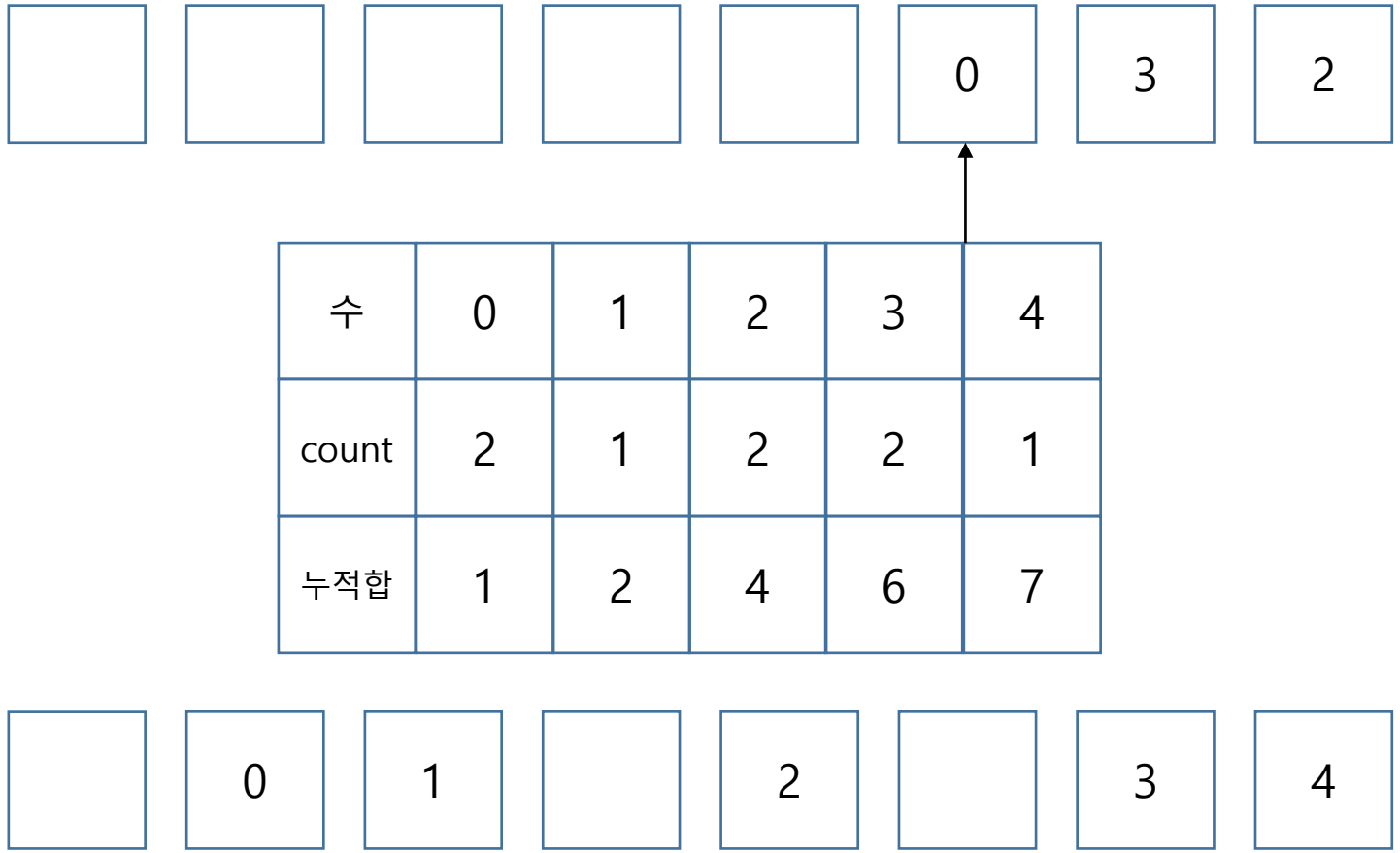
기타

▶ Counting sort



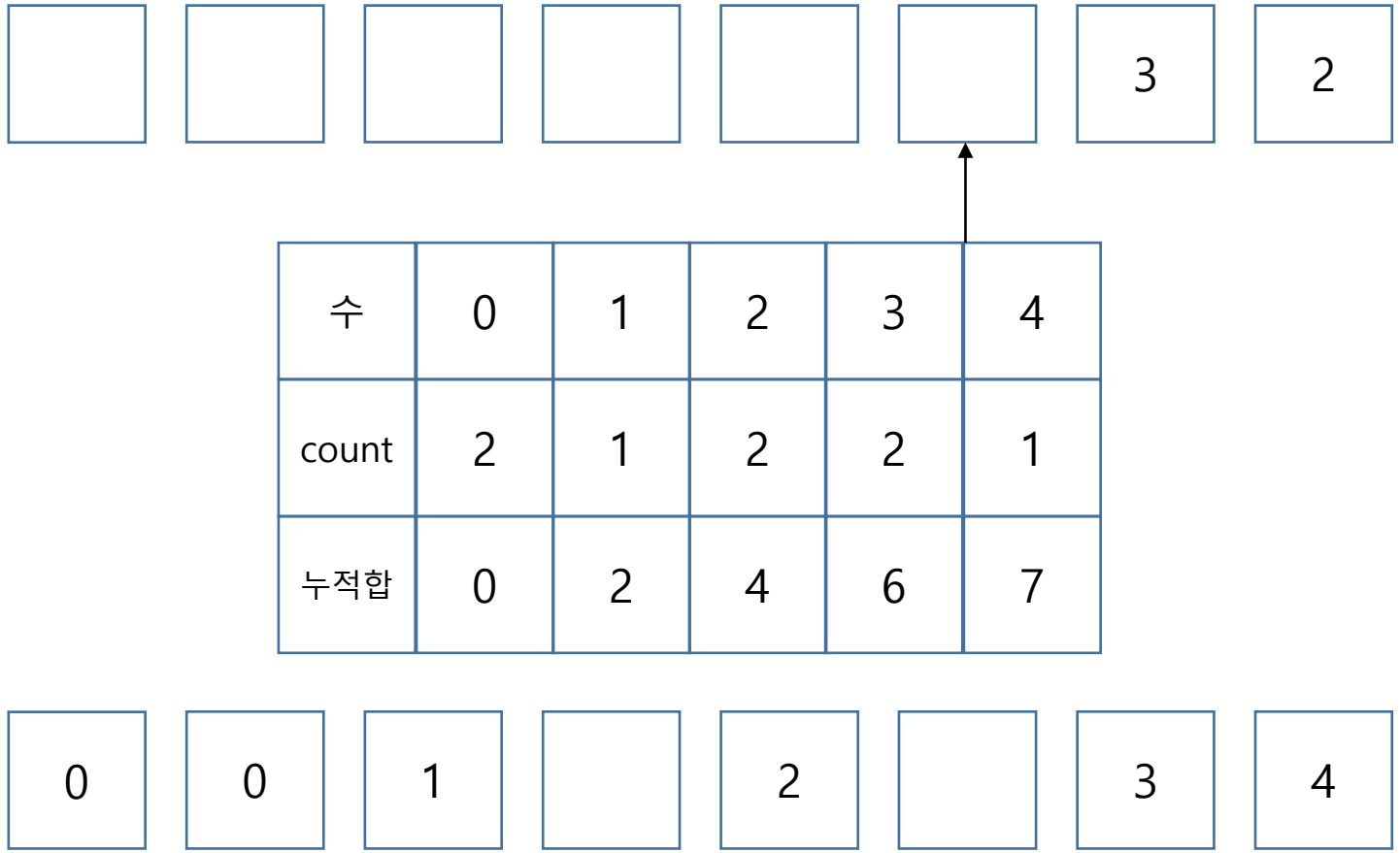
기타

▶ Counting sort



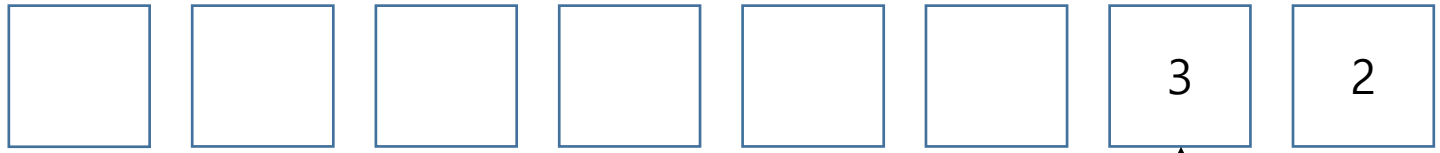
기타

▶ Counting sort

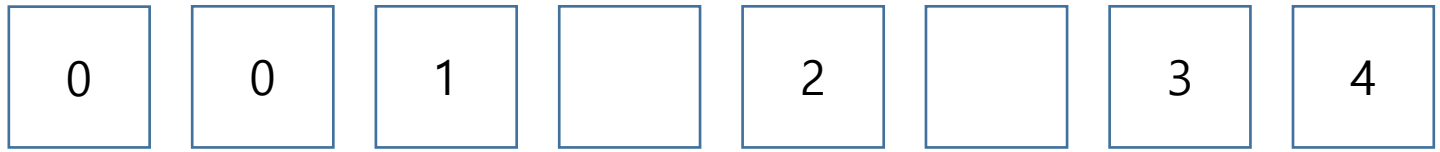


기타

▶ Counting sort

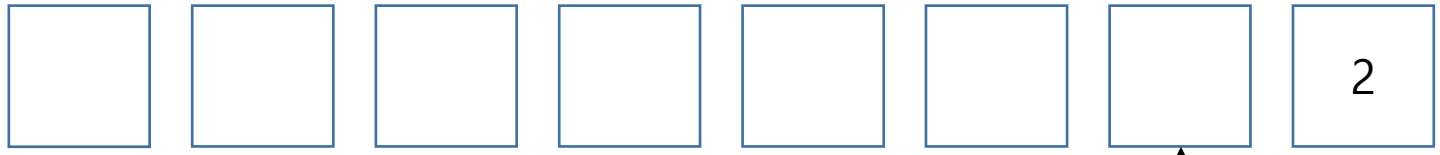


수	0	1	2	3	4
count	2	1	2	2	1
누적합	0	2	4	6	7

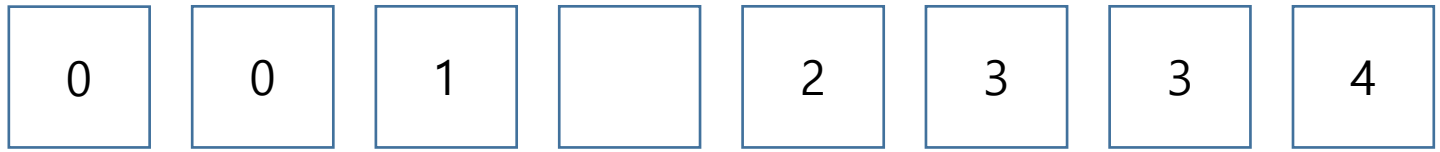


기타

▶ Counting sort

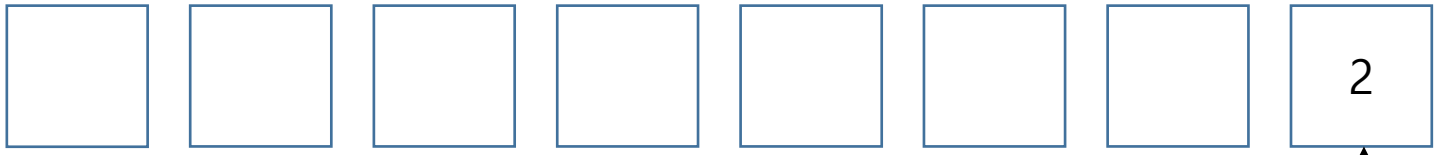


수	0	1	2	3	4
count	2	1	2	2	1
누적합	0	2	4	6	7

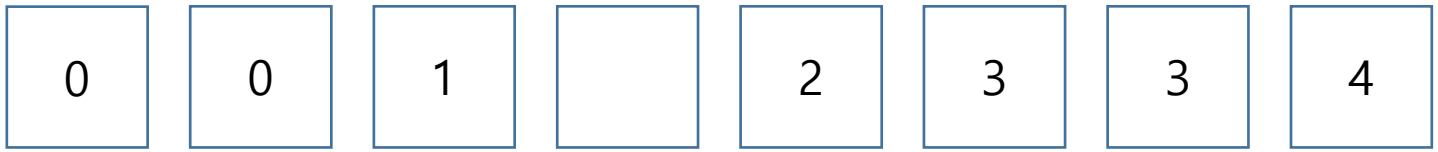


기타

▶ Counting sort

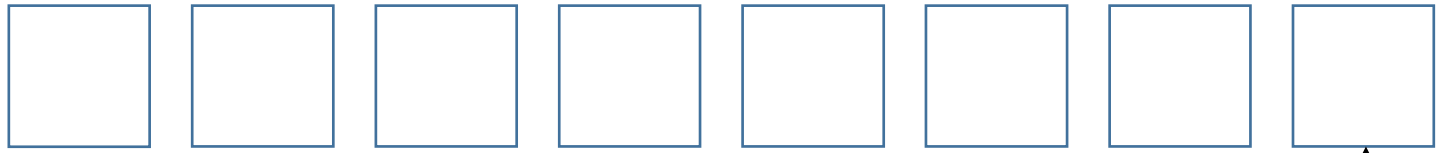


수	0	1	2	3	4
count	2	1	2	2	1
누적합	0	2	4	6	7

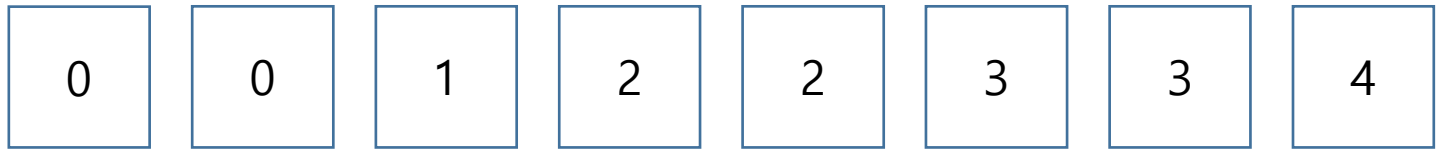


기타

▶ Counting sort



수	0	1	2	3	4
count	2	1	2	2	1
누적합	0	2	4	6	7



기타

- Counting sort

  왜 그냥 count를 이용하지 않는가?

  0 : 2개, 1: 1개, 2: 2개, 3: 2개, 4: 1개 라면 순서대로

  0 0 1 2 2 3 3 4 로 만들면 되는 것 아닌가?

수	0	1	2	3	4
count	2	1	2	2	1

기타

- Counting sort

  숫자만 있는 것이 아닌 데이터도 같이 있는 경우에

  구별할 수 없기 때문이 아닐까?

기타

- Counting sort

  O(k + n) 의 시간 복잡도를 가짐 (k = 최대 값)

  N이 크고 최대값이 작은 경우 nlogn보다 유리 할 수 있음

문제

- 문제번호

  2751 – 수 정렬하기2(라이브러리 함수x)

  10989 – 수 정렬하기3

  10814 – 나이순 정렬

  1377 - 버블 소트

  10800 - 컬러볼

Q & A

감사합니다