# 정렬 알고리즘

알고리즘 스터디 #6

Author: @hheeseung

# 정렬 알고리즘

- 데이터를 특정 기준에 따라 순서대로 나열하는 방식
- 다음 데이터를 어떻게 정렬할 수 있을까?

| 7 | 5 | 9 | 0 | 3 | 1 | 6 | 2 | 4 | 8 |
|---|---|---|---|---|---|---|---|---|---|

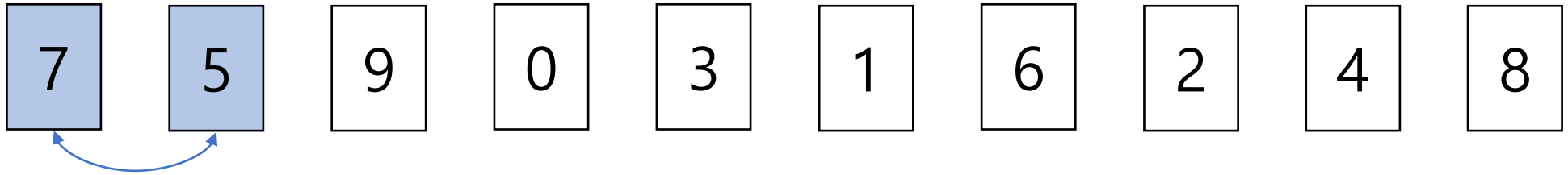
- 오름차순 정렬을 기준으로 살펴볼 것임

# 버블 정렬 (Bubble Sort)

- 옆에 있는 값과 비교해서 더 작은 값을 앞으로 보내는 정렬 알고리즘
- 효율이 가장 떨어지는 방식이라고 함

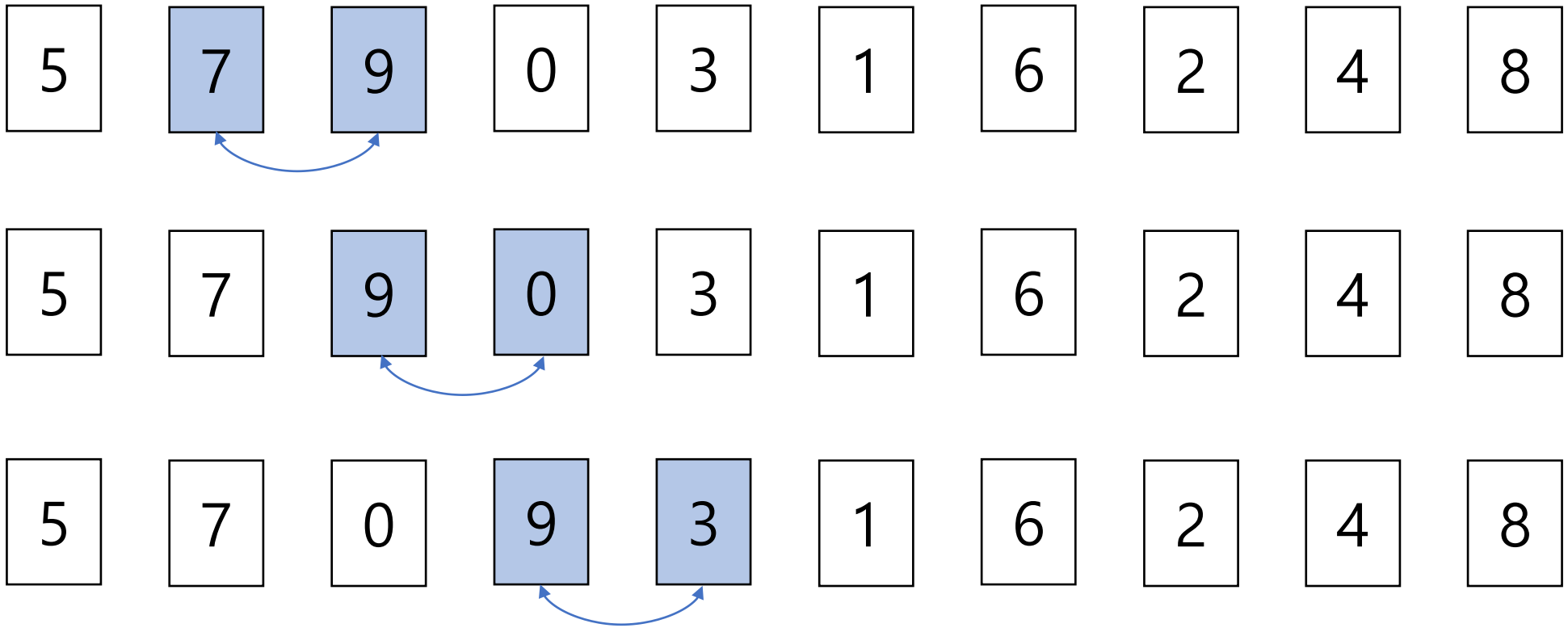
**[Step1]**

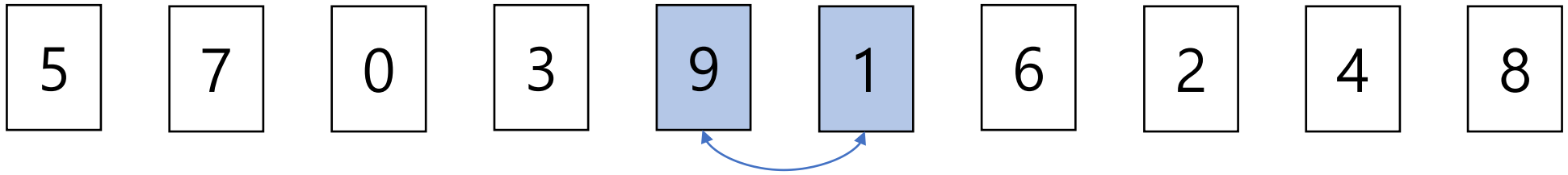
맨 앞의 원소 값을 다음 원소 값과 비교해서 더 작은 것을 앞으로 보냄



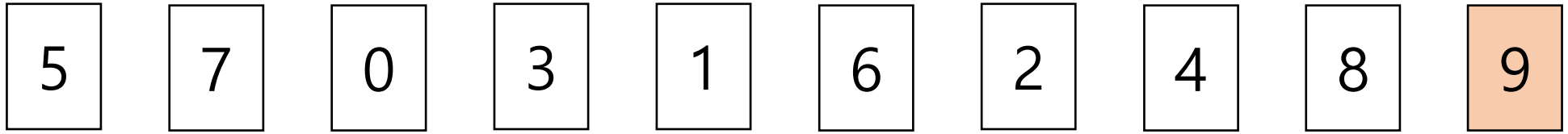
**[Step 2]**

두번째 원소 값을 다음 원소 값과 비교하여 작은 것을 앞으로 보냄






## 1회 버블정렬 결과




→ 1회씩 정렬을 할 때마다 데이터의 가장 큰 값이 맨 뒤로 밀려나는 구조

BubbleSort.java

```java
package week6;

public class BubbleSort {
    public static void main(String[] args) {
        int i, j, temp;
        int[] array = {1, 10, 5, 8, 7, 6, 4, 3, 2, 9};
        for (i = 0; i < array.length; i++) {
            for (j = 0; j < array.length-1; j++) {
                if (array[j] > array[j + 1]) {
                    temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
            }
        }
        for (i = 0; i < array.length; i++) {
            System.out.printf("%d ", array[i]);
        }
    }
}

```

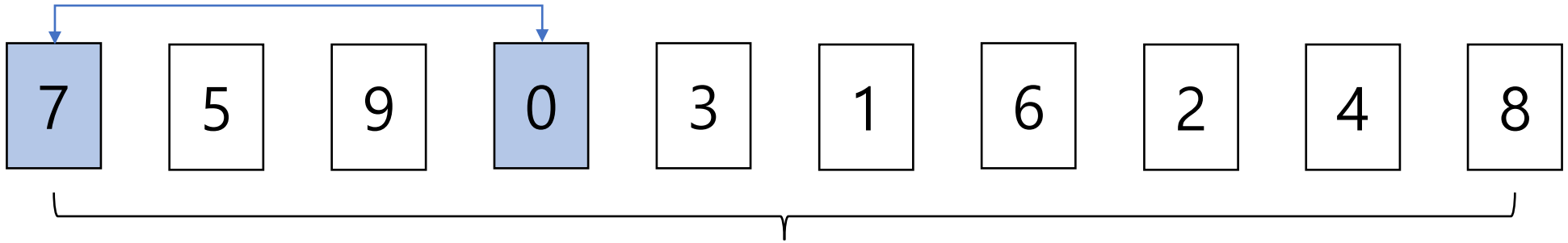
Run: BubbleSort

```
"C:\Program Files\Java\jdk-11.0.12\bin\java.exe"
1 2 3 4 5 6 7 8 9 10
Process finished with exit code 0
```

# 선택 정렬 (Selection Sort)

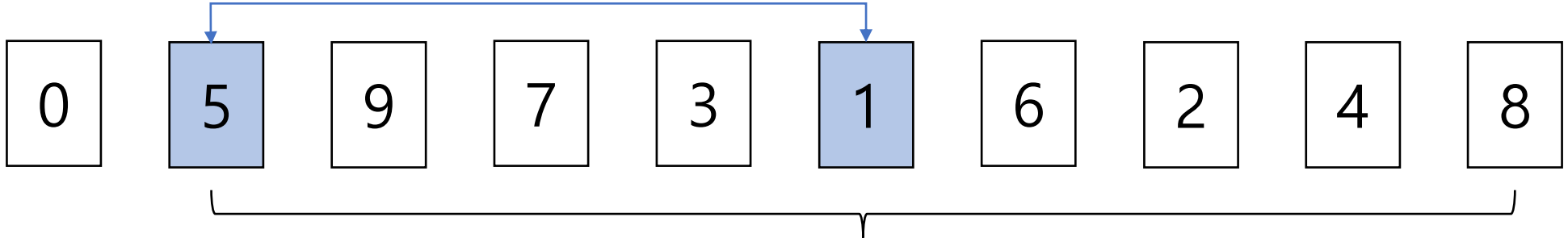
- 처리되지 않은 데이터 중 가장 작은 데이터를 선택하여 맨 앞의 데이터와 바꾸는 것을 반복하는 알고리즘

**[Step 1]**



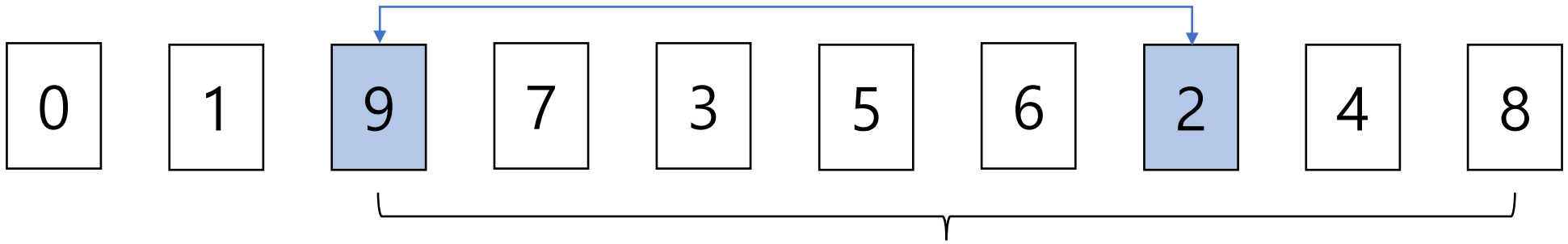

처리되지 않은 데이터 중 가장 작은 '0'을 선택하여 맨 앞의 '7'과 바꿈

**[Step 2]**

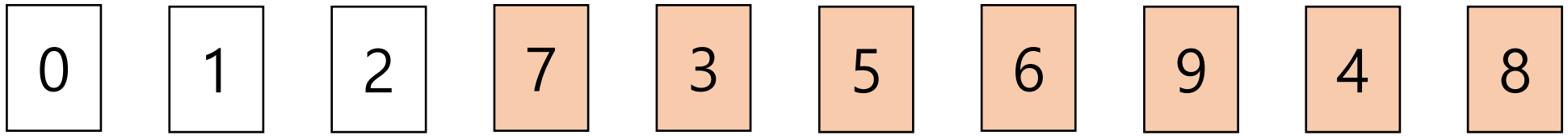



처리되지 않은 데이터 중 가장 작은 '1'을 선택하여 맨 앞의 '5'와 바꿈

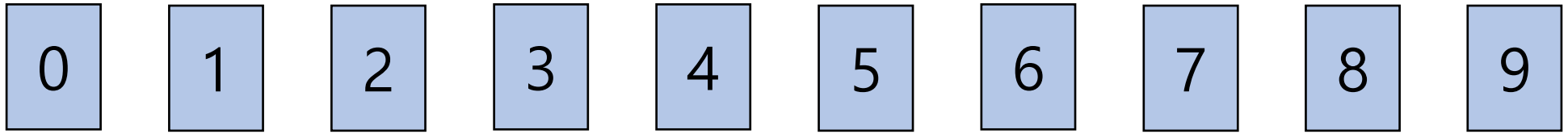
**[Step 3]**




처리되지 않은 데이터 중 가장 작은 '2'를 선택하여 맨 앞의 '9'와 바꿈




이 과정을 반복한 결과:



# 선택 정렬 소스코드 (Java)

```java
import java.util.*;

public class Main {

    public static void main(String[] args) {
        int n = 10;
        int[] arr = {7, 5, 9, 0, 3, 1, 6, 2, 4, 8};

        for (int i = 0; i < n; i++) {
            int min_index = i; // 가장 작은 원소의 인덱스
            for (int j = i + 1; j < n; j++) {
                if (arr[min_index] > arr[j]) {
                    min_index = j;
                }
            }
            // 스와프
            int temp = arr[i];
            arr[i] = arr[min_index];
            arr[min_index] = temp;
        }
        for(int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

실행 결과

```
0 1 2 3 4 5 6 7 8 9
```

# 선택 정렬 소스코드 (Python)

```python
array = [7, 5, 9, 0, 3, 1, 6, 2, 4, 8]

for i in range(len(array)):
    min_index = i # 가장 작은 원소의 인덱스
    for j in range(i + 1, len(array)):
        if array[min_index] > array[j]:
            min_index = j
    array[i], array[min_index] = array[min_index], array[i] # 스와프

print(array)
```

실행 결과

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# 선택 정렬 소스코드 (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

int n = 10;
int target[10] = {7, 5, 9, 0, 3, 1, 6, 2, 4, 8};

int main(void) {
    for (int i = 0; i < n; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) {
            if (target[min_index] > target[j]) {
                min_index = j;
            }
        }
        swap(target[i], target[min_index]);
    }
    for(int i = 0; i < n; i++) {
        cout << target[i] << ' ';
    }
    return 0;
}
```

실행 결과

```
0 1 2 3 4 5 6 7 8 9
```

# 삽입 정렬 (Insertion Sort)

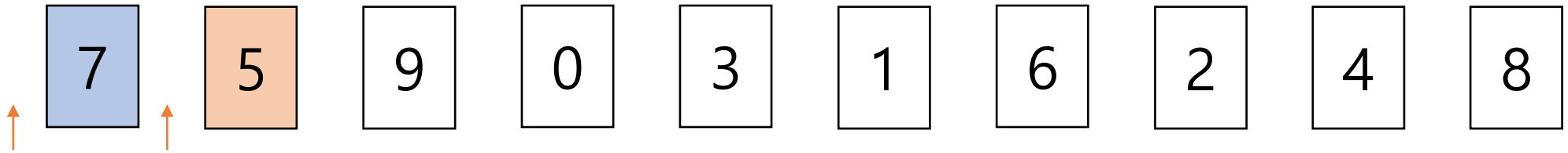
- 처리되지 않은 데이터를 하나씩 골라 적절한 위치에 삽입하는 정렬
- 선택 정렬보다 효율적
- 현재 리스트의 데이터가 거의 정렬된 상태다 → 매우 빠르게 동작

- 앞쪽의 원소들이 이미 정렬되어 있다고 가정하고 뒤쪽의 원소를 앞쪽 원소를 기준으로 어느 위치에 넣을지 판단함
- 넣을 위치: 앞쪽 데이터의 왼쪽 or 오른쪽

**[Step 1]**

첫번째 데이터 '7' → 정렬된 걸로 간주

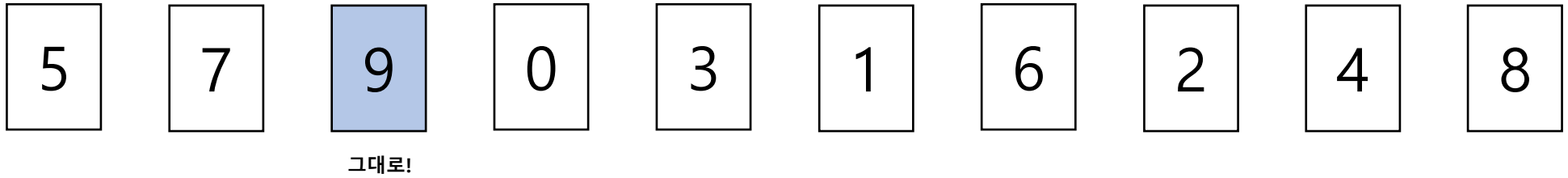
두번째 데이터 '5'가 어느 위치로 갈지...?



**[Step 2]**

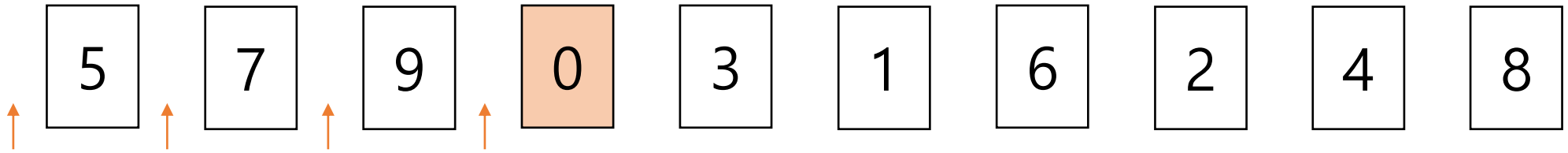
이어서 '9'가 어느 위치로 들어갈지 판단:

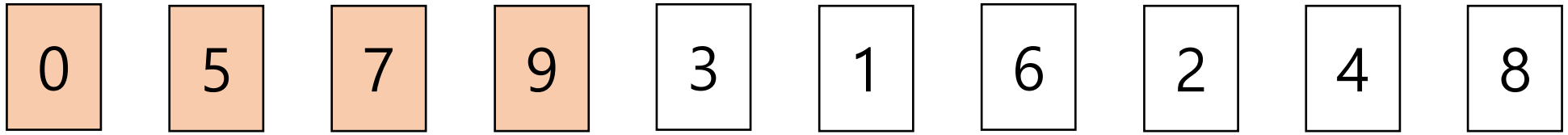
앞의 데이터 5, 7보다 크기 때문에 그 자리에 STAY



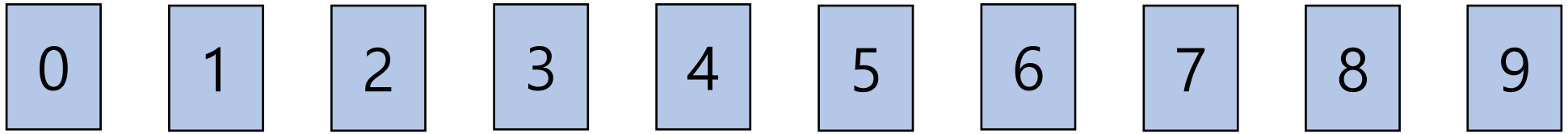

**[Step 3]**

이어서 '0'이 어느 위치로 들어갈지 판단






이 과정을 반복한 결과:



# 삽입 정렬 소스코드 (Java)

```java
import java.util.*;

public class Main {

    public static void main(String[] args) {
        int n = 10;
        int[] arr = {7, 5, 9, 0, 3, 1, 6, 2, 4, 8};
        for (int i = 1; i < n; i++) {
            // 인덱스 i부터 1까지 감소하며 반복하는 문법
            for (int j = i; j > 0; j--) {
                // 한 칸씩 왼쪽으로 이동
                if (arr[j] < arr[j - 1]) {
                    // 스와프(Swap)
                    int temp = arr[j];
                    arr[j] = arr[j - 1];
                    arr[j - 1] = temp;
                }
                // 자기보다 작은 데이터를 만나면 그 위치에서 멈춤
                else break;
            }
        }
        for(int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

실행 결과

```
0 1 2 3 4 5 6 7 8 9
```

# 삽입 정렬 소스코드 (Python)

```python
array = [7, 5, 9, 0, 3, 1, 6, 2, 4, 8]

for i in range(1, len(array)):
    for j in range(i, 0, -1): # 인덱스 i부터 1까지 1씩 감소하며 반복하는 문법
        if array[j] < array[j - 1]: # 한 칸씩 왼쪽으로 이동
            array[j], array[j - 1] = array[j - 1], array[j]
        else: # 자기보다 작은 데이터를 만나면 그 위치에서 멈춤
            break

print(array)
```

실행 결과

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# 삽입 정렬 소스코드 (C++)

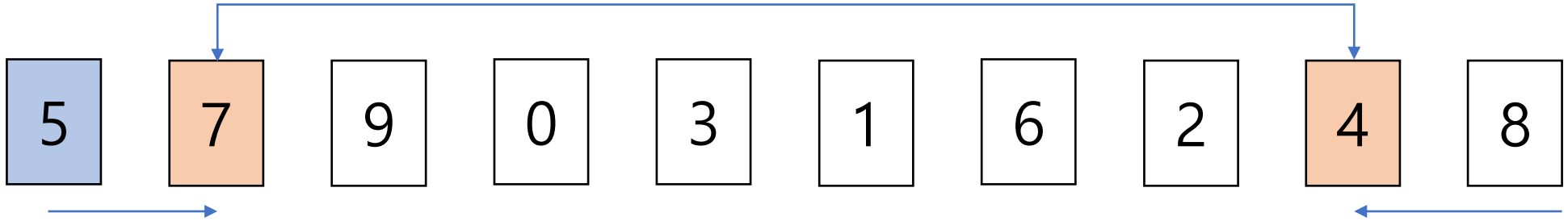
```cpp
#include <bits/stdc++.h>

using namespace std;

int n = 10;
int target[10] = {7, 5, 9, 0, 3, 1, 6, 2, 4, 8};

int main(void) {
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0; j--) {
            if (target[j] < target[j - 1]) {
                swap(target[j], target[j - 1]);
            }
            else break;
        }
    }
    for(int i = 0; i < n; i++) {
        cout << target[i] << ' ';
    }
    return 0;
}
```

실행 결과

```
0 1 2 3 4 5 6 7 8 9
```

# 퀵 정렬 (Quick Sort)

- 굉장히 빠른 정렬 알고리즘 중 하나
- 기준 데이터(=Pivot)를 정하고 그 기준보다 큰 데이터와 작은 데이터의 위치를 바꾸는 방법
- 일반적인 상황에서 가장 많이 사용되는 알고리즘 중 하나
- 가장 기본적인 정렬: 첫번째 데이터를 피벗으로 설정



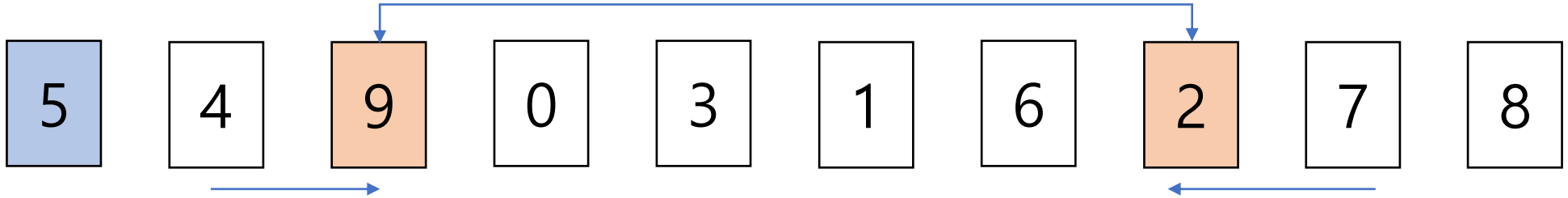

**[Step 1]**

현재 피벗 값은 '5'

왼쪽: 피벗보다 큰 값 = '5'보다 큰 데이터인 '7' 선택

오른쪽: 피벗보다 작은 값 = '5'보다 작은 데이터인 '4' 선택

'7'과 '4'의 위치를 서로 변경



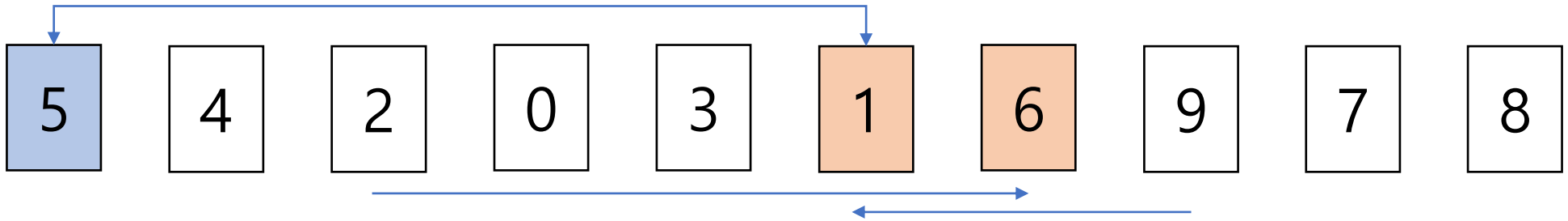

**[Step 2]**

현재 피벗 값은 '5'

왼쪽: 피벗보다 큰 값 = '5'보다 큰 데이터인 '9' 선택

오른쪽: 피벗보다 작은 값 = '5'보다 작은 데이터인 '2' 선택

'9'와 '2'의 위치를 서로 변경



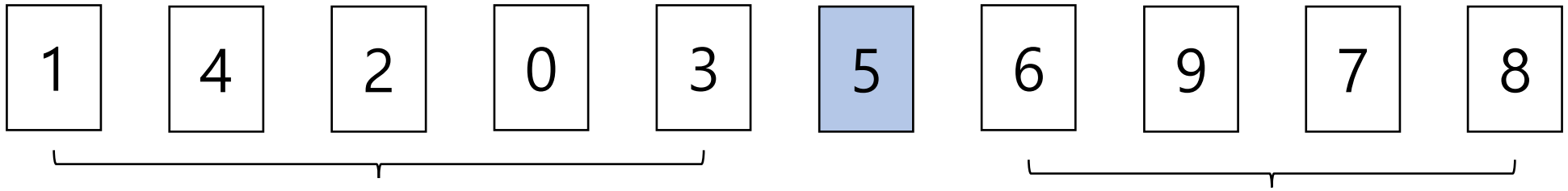

**[Step 3]**

현재 피벗 값은 '5'

왼쪽: 피벗보다 큰 값 = '5'보다 큰 데이터인 '6' 선택

오른쪽: 피벗보다 작은 값 = '5'보다 작은 데이터인 '1' 선택

현재처럼 위치가 엇갈릴 경우 : 피벗과 작은 데이터의 위치를 서로 변경



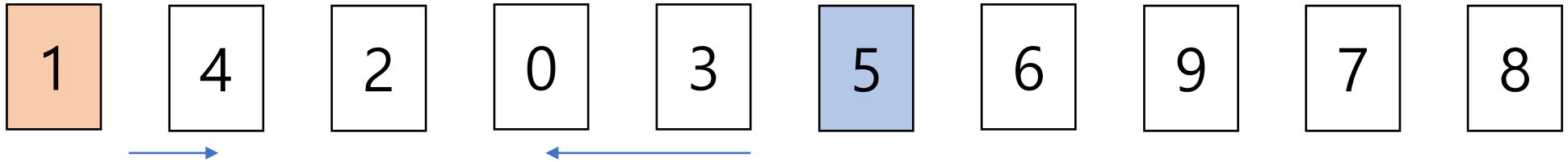

**[분할 완료]**

피벗 ‘5’를 기준으로 왼쪽에 피벗보다 작은 데이터, 오른쪽에 피벗보다 큰 데이터가 현재 정렬됨

**분할(Divide)** : 피벗을 기준으로 데이터 묶음을 나누는 작업

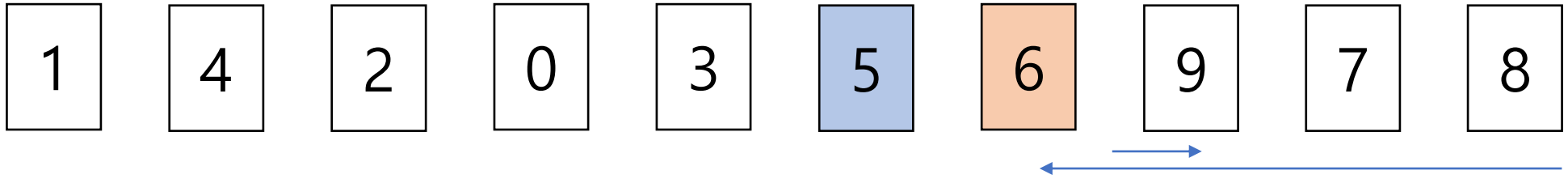
**[왼쪽 데이터 묶음 정렬]**

피벗 5를 기준으로 왼쪽에 있는 데이터에 대해 퀵정렬을 반복적으로 수행




**[오른쪽 데이터 묶음 정렬]**

피벗 5를 기준으로 오른쪽에 있는 데이터에 대해 퀵정렬을 반복적으로 수행



```java
import java.util.*;

public class Main {

    public static void quickSort(int[] arr, int start, int end) {
        if (start >= end) return; // 원소가 1개인 경우 종료
        int pivot = start; // 피벗은 첫 번째 원소
        int left = start + 1;
        int right = end;
        while (left <= right) {
            // 피벗보다 큰 데이터를 찾을 때까지 반복
            while (left <= end && arr[left] <= arr[pivot]) left++;
            // 피벗보다 작은 데이터를 찾을 때까지 반복
            while (right > start && arr[right] >= arr[pivot]) right--;
            // 엇갈렸다면 작은 데이터와 피벗을 교체
            if (left > right) {
                int temp = arr[pivot];
                arr[pivot] = arr[right];
                arr[right] = temp;
            }
            // 엇갈리지 않았다면 작은 데이터와 큰 데이터를 교체
            else {
                int temp = arr[left];
                arr[left] = arr[right];
                arr[right] = temp;
            }
        }
        // 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬 수행
        quickSort(arr, start, right - 1);
        quickSort(arr, right + 1, end);
    }

    public static void main(String[] args) {
        int n = 10;
        int[] arr = {7, 5, 9, 0, 3, 1, 6, 2, 4, 8};

        quickSort(arr, 0, n - 1);

        for(int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
    }

}
```

# 퀵 정렬 소스코드: 일반적인 방식 (Python)

```python
array = [5, 7, 9, 0, 3, 1, 6, 2, 4, 8]

def quick_sort(array, start, end):
    if start >= end: # 원소가 1개인 경우 종료
        return
    pivot = start # 피벗은 첫 번째 원소
    left = start + 1
    right = end
    while(left <= right):
        # 피벗보다 큰 데이터를 찾을 때까지 반복
        while(left <= end and array[left] <= array[pivot]):
            left += 1
        # 피벗보다 작은 데이터를 찾을 때까지 반복
        while(right > start and array[right] >= array[pivot]):
            right -= 1
        if(left > right): # 엇갈렸다면 작은 데이터와 피벗을 교체
            array[right], array[pivot] = array[pivot], array[right]
        else: # 엇갈리지 않았다면 작은 데이터와 큰 데이터를 교체
            array[left], array[right] = array[right], array[left]
    # 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬 수행
    quick_sort(array, start, right - 1)
    quick_sort(array, right + 1, end)

quick_sort(array, 0, len(array) - 1)
print(array)
```

실행 결과

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# 퀵 정렬 소스코드: 일반적인 방식 (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

int n = 10;
int target[10] = {7, 5, 9, 0, 3, 1, 6, 2, 4, 8};

void quickSort(int* target, int start, int end) {
    if (start >= end) return;
    int pivot = start;
    int left = start + 1;
    int right = end;
    while (left <= right) {
        while (left <= end && target[left] <= target[pivot]) left++;
        while (right > start && target[right] >= target[pivot]) right--;
        if (left > right) swap(target[pivot], target[right]);
        else swap(target[left], target[right]);
    }
    quickSort(target, start, right - 1);
    quickSort(target, right + 1, end);
}

int main(void) {
    quickSort(target, 0, n - 1);
    for (int i = 0; i < n; i++) cout << target[i] << ' ';
    return 0;
}
```

실행 결과

```
0 1 2 3 4 5 6 7 8 9
```

# 퀵 정렬 소스코드: 파이썬의 장점을 살린 방식

```python
array = [5, 7, 9, 0, 3, 1, 6, 2, 4, 8]

def quick_sort(array):
    # 리스트가 하나 이하의 원소만을 담고 있다면 종료
    if len(array) <= 1:
        return array
    pivot = array[0] # 피벗은 첫 번째 원소
    tail = array[1:] # 피벗을 제외한 리스트

    left_side = [x for x in tail if x <= pivot] # 분할된 왼쪽 부분
    right_side = [x for x in tail if x > pivot] # 분할된 오른쪽 부분

    # 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬 수행하고, 전체 리스트 반환
    return quick_sort(left_side) + [pivot] + quick_sort(right_side)

print(quick_sort(array))
```

실행 결과

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# 계수 정렬 (Counting Sort)

- 특정 조건이 부합할 때만 사용할 수 있지만 매우 빠르게 동작
- 계수 정렬: 데이터의 크기범위가 제한되어 정수 형태로 표현할 수 있을 때 사용 가능

## [Step 1]

- 가장 작은 데이터~가장 큰 데이터까지의 범위가 모두 담길 수 있는 리스트 생성
- 정렬할 데이터: 7 5 9 0 3 1 6 2 9 1 4 8 0 5 2

| **Index** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Count** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Index = 데이터의 값

Count(개수) = 각각의 데이터가 총 몇 번씩 등장했는지 그 횟수

## [Step 2]

- 데이터를 하나씩 확인하며 데이터 값과 동일한 인덱스의 데이터 1씩 증가
- 정렬할 데이터: 7 5 9 0 3 1 6 2 9 1 4 8 0 5 2

| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 개수(Count) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

## [Step 3]

- 데이터를 하나씩 확인하며 데이터 값과 동일한 인덱스의 데이터 1씩 증가
- 정렬할 데이터: 7 5 9 0 3 1 6 2 9 1 4 8 0 5 2

| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 개수(Count) | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

## [Step 4]

- 데이터를 하나씩 확인하며 데이터 값과 동일한 인덱스의 데이터 1씩 증가
- 정렬할 데이터: 7 5 9 0 3 1 6 2 9 1 4 8 0 5 2

| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 개수(Count) | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

## .....[Step 16]

- 데이터를 하나씩 확인하며 데이터 값과 동일한 인덱스의 데이터 1씩 증가
- 정렬할 데이터: 7 5 9 0 3 1 6 2 9 1 4 8 0 5 2

| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 개수(Count) | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 2 |

- 결과 확인 시 리스트의 첫번째 데이터부터 하나씩 그 값만큼 반복하여 인덱스 출력

| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 개수(Count) | 2 | 2 | 2 | 1 | 1 | 2 | 1 | 1 | 1 | 2 |

- **출력 결과:** 0 0 1 1 2 2 3 4 5 5 6 7 8 9 9

# 계수 정렬 소스코드 (Java)

```java
import java.util.*;

public class Main {
    public static final int MAX_VALUE = 9;

    public static void main(String[] args) {
        int n = 15;
        // 모든 원소의 값이 0보다 크거나 같다고 가정
        int[] arr = {7, 5, 9, 0, 3, 1, 6, 2, 9, 1, 4, 8, 0, 5, 2};
        // 모든 범위를 포함하는 배열 선언(모든 값은 0으로 초기화)
        int[] cnt = new int[MAX_VALUE + 1];

        for (int i = 0; i < n; i++) {
            cnt[arr[i]] += 1; // 각 데이터에 해당하는 인덱스의 값 증가
        }
        for (int i = 0; i <= MAX_VALUE; i++) { // 배열에 기록된 정렬 정보 확인
            for (int j = 0; j < cnt[i]; j++) {
                System.out.print(i + " "); // 띄어쓰기를 기준으로 등장한 횟수만큼 인덱스 출력
            }
        }
    }
}
```

실행 결과

```
0 0 1 1 2 2 3 4 5 5 6 7 8 9 9
```

# 계수 정렬 소스코드 (Python)

```python
# 모든 원소의 값이 0보다 크거나 같다고 가정
array = [7, 5, 9, 0, 3, 1, 6, 2, 9, 1, 4, 8, 0, 5, 2]
# 모든 범위를 포함하는 리스트 선언(모든 값은 0으로 초기화)
count = [0] * (max(array) + 1)

for i in range(len(array)):
    count[array[i]] += 1 # 각 데이터에 해당하는 인덱스의 값 증가

for i in range(len(count)): # 리스트에 기록된 정렬 정보 확인
    for j in range(count[i]):
        print(i, end=' ') # 띄어쓰기를 구분으로 등장한 횟수만큼 인덱스 출력
```

실행 결과

```
0 0 1 1 2 2 3 4 5 5 6 7 8 9 9
```

# 계수 정렬 소스코드 (C++)

```cpp
#include <bits/stdc++.h>
#define MAX_VALUE 9

using namespace std;

int n = 15;
// 모든 원소의 값이 0보다 크거나 같다고 가정
int arr[15] = {7, 5, 9, 0, 3, 1, 6, 2, 9, 1, 4, 8, 0, 5, 2};
// 모든 범위를 포함하는 배열 선언(모든 값은 0으로 초기화)
int cnt[MAX_VALUE + 1];

int main(void) {
    for (int i = 0; i < n; i++) {
        cnt[arr[i]] += 1; // 각 데이터에 해당하는 인덱스의 값 증가
    }
    for (int i = 0; i <= MAX_VALUE; i++) { // 배열에 기록된 정렬 정보 확인
        for (int j = 0; j < cnt[i]; j++) {
            cout << i << ' '; // 띄어쓰기를 기준으로 등장한 횟수만큼 인덱스 출력
        }
    }
}
```

실행 결과

```
0 0 1 1 2 2 3 4 5 5 6 7 8 9 9
```

# 계수 정렬의 효율성

- 계수 정렬은 때에 따라서 심각한 비효율성을 초래할 수 있음

ex)  데이터가 0과 999,999로 단 2개만 존재하는 경우

- 동일한 값을 갖는 데이터가 여러 개 등장할 때 효과적으로 사용가능

ex) 성적의 경우 100점을 맞은 학생이 여러 명일 수 있기 때문에 계수 정렬이 효과적임

# 풀어올 문제

https://www.acmicpc.net/problem/2751