

창의적 문제 해결

7주차~9주차

공주대학교 컴퓨터공학과

김영부

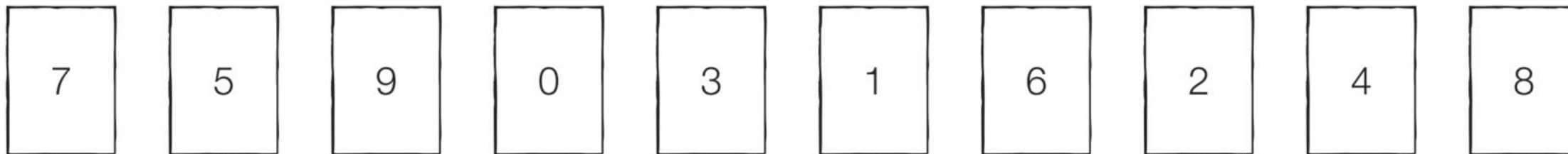


- 정렬 알고리즘
 - 선택 정렬
 - 삽입 정렬
 - 퀵 정렬
 - 계수 정렬
 - 정렬 알고리즘 비교
- 이진 탐색 알고리즘 및 문제 풀이

정렬 알고리즘: 선택 정렬

정렬 알고리즘

- 정렬(Sorting)이란 데이터를 특정한 기준에 따라 순서대로 나열하는 것을 말합니다.
- 일반적으로 문제 상황에 따라서 적절한 정렬 알고리즘이 공식처럼 사용됩니다.



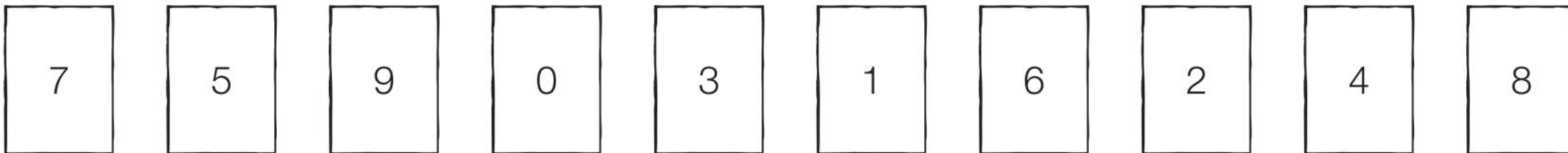
여러 개의 데이터(카드)를 어떻게 정렬할 수 있을까요?

선택 정렬

- 처리되지 않은 데이터 중에서 가장 작은 데이터를 **선택**해 맨 앞에 있는 데이터와 바꾸는 것을 반복합니다(오름차순 정렬 기준).

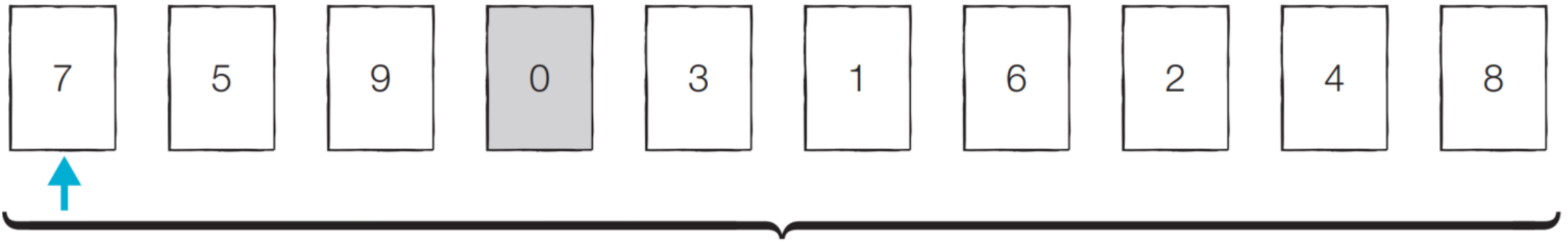
선택 정렬 동작 예시

- 정렬할 데이터를 준비합니다.



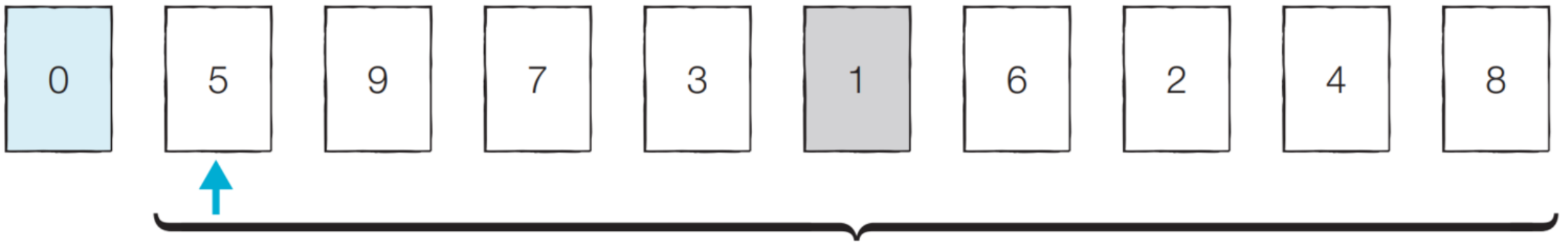
선택 정렬 동작 예시

- [Step 0] 처리되지 않은 데이터 중 가장 작은 '0'을 선택해 가장 앞의 '7'과 바꿉니다.



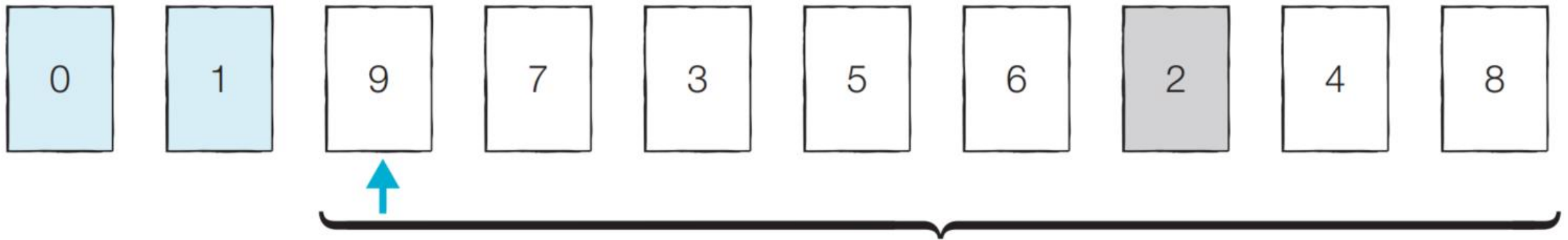
선택 정렬 동작 예시

- [Step 1] 처리되지 않은 데이터 중 가장 작은 '1'을 선택해 가장 앞의 '5'와 바꿉니다.



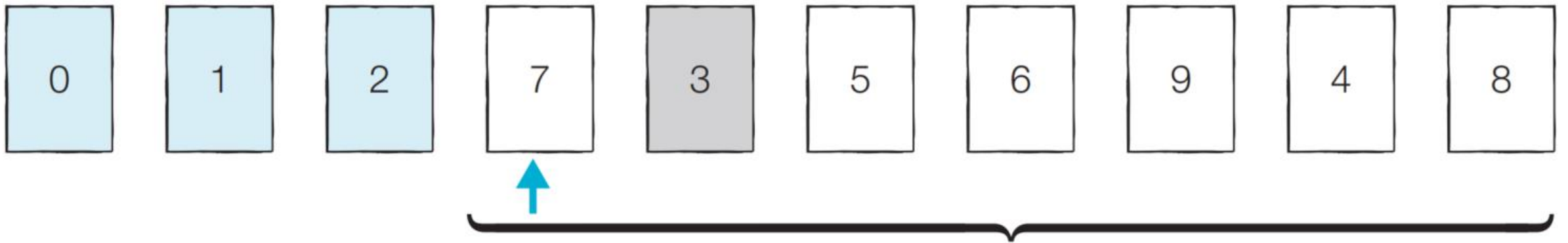
선택 정렬 동작 예시

- [Step 2] 처리되지 않은 데이터 중 가장 작은 '2'를 선택해 가장 앞의 '9'와 바꿉니다.



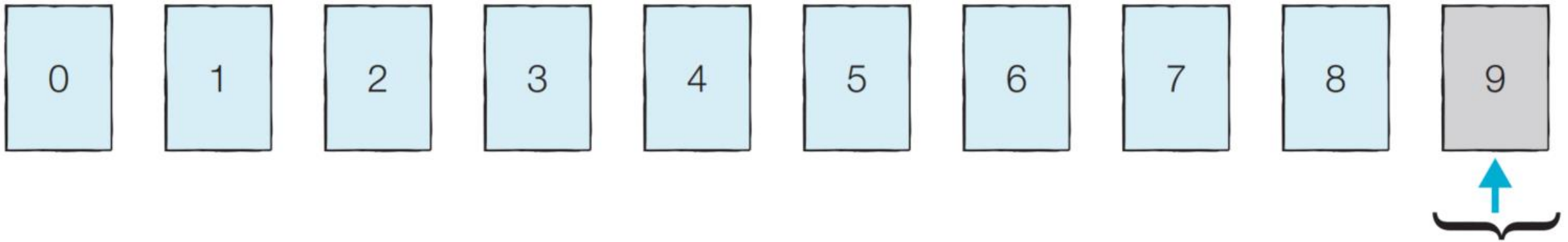
선택 정렬 동작 예시

- [Step 3] 처리되지 않은 데이터 중 가장 작은 '3'을 선택해 가장 앞의 '7'과 바꿉니다.



선택 정렬 동작 예시

- 이러한 과정을 반복하면 다음과 같이 정렬이 완료됩니다.



선택 정렬 소스코드 (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++)

```
#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

선택 정렬 소스코드 (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 정렬의 시간 복잡도

- 선택 정렬은 N번 만큼 가장 작은 수를 찾아서 맨 앞으로 보내야 합니다.
- 구현 방식에 따라서 사소한 오차는 있을 수 있지만, 전체 연산 횟수는 다음과 같습니다.

$$N + (N - 1) + (N - 2) + \dots + 2$$

- 이는 $(N^2 + N - 2) / 2$ 로 표현할 수 있는데, 빅오 표기법에 따라서 $O(N^2)$ 이라고 작성합니다.

데이터의 개수(N)	선택 정렬	퀵 정렬	기본 정렬 라이브러리
N = 100	0.0123초	0.00156초	0.00000753초
N = 1,000	0.354초	0.00343초	0.0000365초
N = 10,000	15.475초	0.0312초	0.000248초

선택 정렬은 기본 정렬 라이브러리를 포함해 뒤에서 다룬 알고리즘과 비교했을 때 매우 비효율적이다. 다만, 특정한 리스트에서 가장 작은 데이터를 찾는 일이 코딩 테스트에서 잦으므로 선택 정렬 소스코드 형태에 익숙해질 필요가 있다. 그러므로 선택 정렬 소스코드를 자주 작성해볼 것을 권한다.

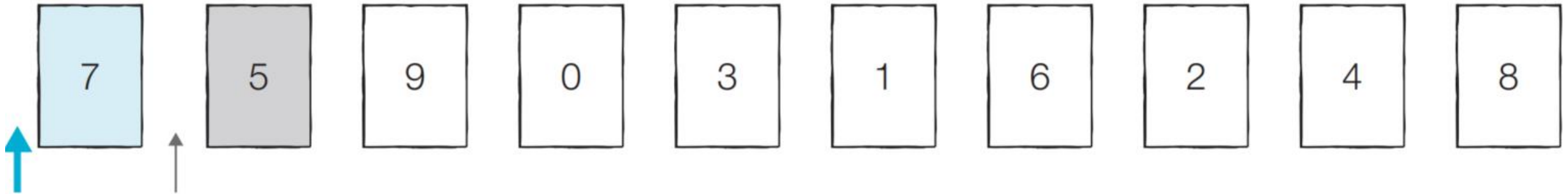
정렬 알고리즘: 삽입 정렬

삽입 정렬

- 처리되지 않은 데이터를 하나씩 골라 적절한 위치에 **삽입**합니다.
- 선택 정렬에 비해 구현 난이도가 높은 편이지만, 일반적으로 더 효율적으로 동작합니다.

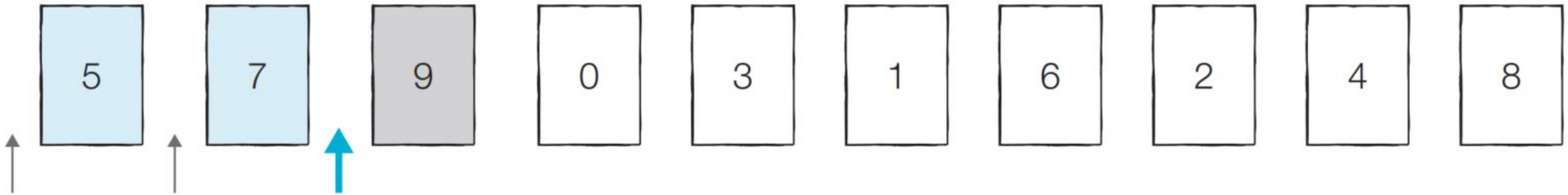
삽입 정렬 동작 예시

- [Step 0] 첫 번째 데이터 '7'은 그 자체로 정렬이 되어 있다고 판단하고, 두 번째 데이터인 '5'가 어떤 위치로 들어갈지 판단합니다. '7'의 왼쪽으로 들어가거나 오른쪽으로 들어가거나 두 경우만 존재합니다.



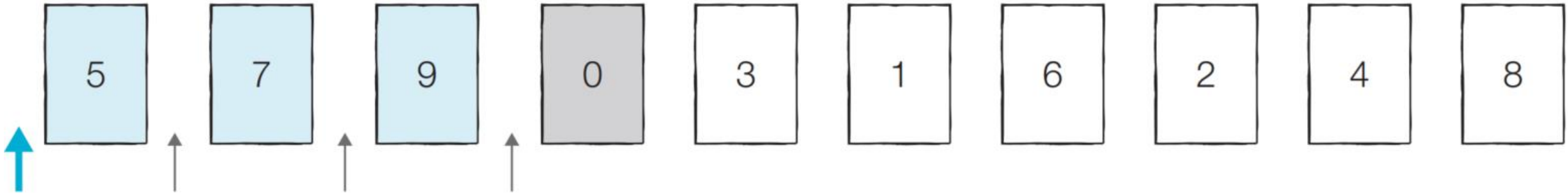
삽입 정렬 동작 예시

- [Step 1] 이어서 '9'가 어떤 위치로 들어갈지 판단합니다.



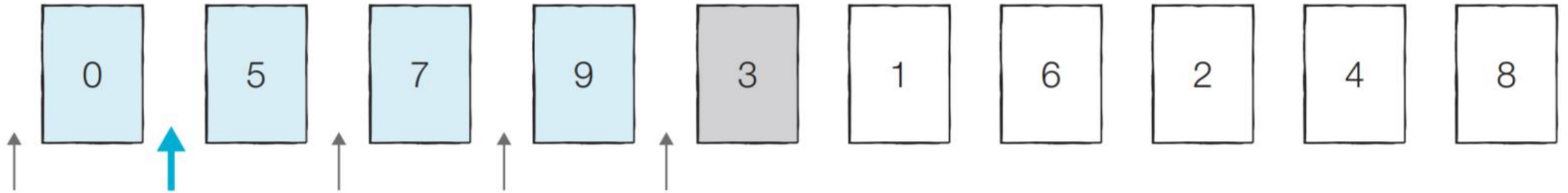
삽입 정렬 동작 예시

- [Step 2] 이어서 '0'이 어떤 위치로 들어갈지 판단합니다.



삽입 정렬 동작 예시

- [Step 3] 이어서 '3'이 어떤 위치로 들어갈지 판단합니다.



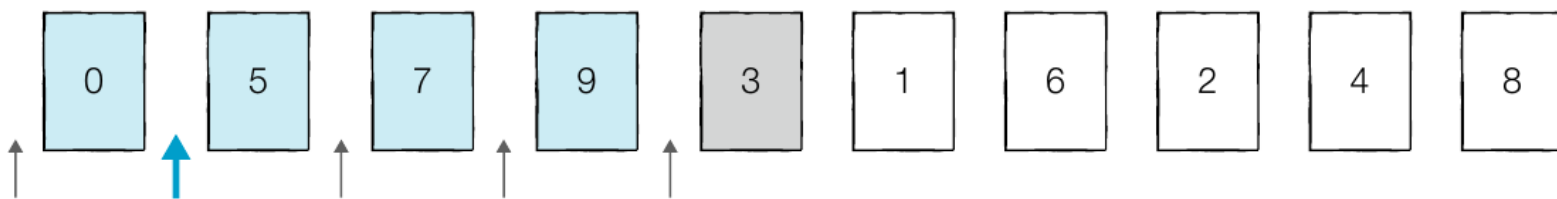
삽입 정렬 동작 예시

- 이러한 과정을 반복하면 다음과 같이 정렬이 완료됩니다.



삽입 정렬 동작 예시

삽입 정렬은 재미있는 특징이 있는데, 정렬이 이루어진 원소는 항상 오름차순을 유지하고 있다는 점이다. [step] 그림을 보면 하늘색으로 칠해진 카드들은 어떤 단계든지 항상 정렬된 상태다. 이러한 특징 때문에 삽입 정렬에서는 특정한 데이터가 삽입될 위치를 선정할 때(삽입될 위치를 찾기 위하여 왼쪽으로 한 칸씩 이동할 때), 삽입될 데이터보다 작은 데이터를 만나면 그 위치에서 멈추면 된다. 예를 들어 [step 3]을 다시 살펴보자.



[step 3]에서 '3'은 한 칸씩 왼쪽으로 이동하다가 자신보다 작은 '0'을 만났을 때 그 위치에 삽입된다. 다시 말해 특정한 데이터의 왼쪽에 있는 데이터들은 이미 정렬이 된 상태이므로 자기보다 작은 데이터를 만났다면 더 이상 데이터를 살펴볼 필요 없이 그 자리에 삽입되면 되는 것이다. 이제 소스코드를 확인해보도록 하자.

삽입 정렬 소스코드 (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++)

```
#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

삽입 정렬 소스코드 (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]) {
                    // 스왑(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 정렬의 시간 복잡도

- 삽입 정렬의 시간 복잡도는 $O(N^2)$ 이며, 선택 정렬과 마찬가지로 반복문이 두 번 중첩되어 사용 됩니다.
- 삽입 정렬은 현재 리스트의 데이터가 거의 정렬되어 있는 상태라면 매우 빠르게 동작합니다.
 - 최선의 경우 $O(N)$ 의 시간 복잡도를 가집니다.
 - 이미 정렬되어 있는 상태에서 다시 삽입 정렬을 수행하면 어떻게 될까요?

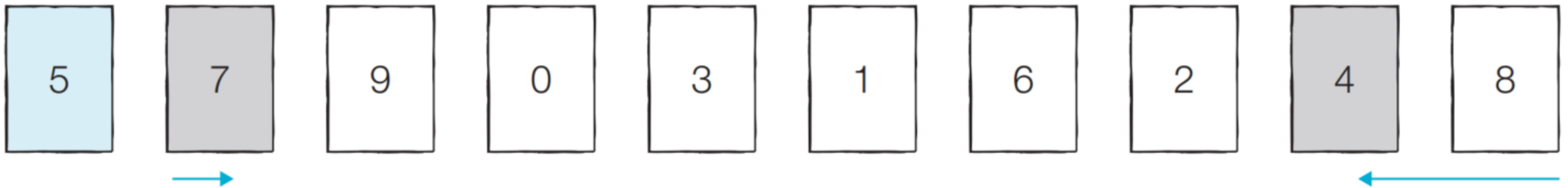
정렬 알고리즘: 퀵 정렬

퀵 정렬

- 기준 데이터를 설정하고 그 기준보다 큰 데이터와 작은 데이터의 위치를 바꾸는 방법입니다.
- 일반적인 상황에서 가장 많이 사용되는 정렬 알고리즘 중 하나입니다.
- 병합 정렬과 더불어 대부분의 프로그래밍 언어의 정렬 라이브러리의 근간이 되는 알고리즘입니다.
- 가장 기본적인 퀵 정렬은 첫 번째 데이터를 기준 데이터(Pivot)로 설정합니다.

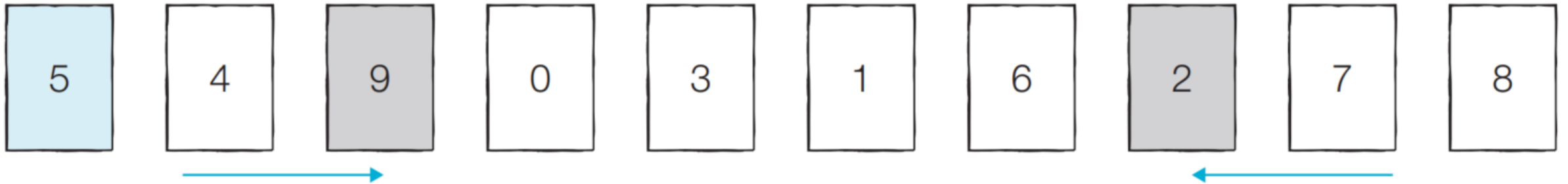
퀵 정렬 동작 예시

- [Step 0] 현재 피벗의 값은 '5'입니다. 왼쪽에서부터 '5'보다 큰 데이터를 선택하므로 '7'이 선택되고, 오른쪽에서부터 '5'보다 작은 데이터를 선택하므로 '4'가 선택됩니다. 이제 이 두 데이터의 위치를 서로 변경합니다.



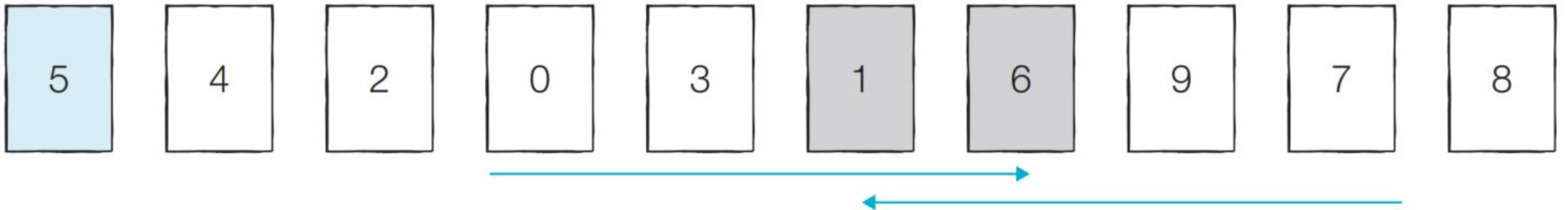
퀵 정렬 동작 예시

- [Step 1] 현재 피벗의 값은 '5'입니다. 왼쪽에서부터 '5'보다 큰 데이터를 선택하므로 '9'가 선택되고, 오른쪽에서부터 '5'보다 작은 데이터를 선택하므로 '2'가 선택됩니다. 이제 이 두 데이터의 위치를 서로 변경합니다.



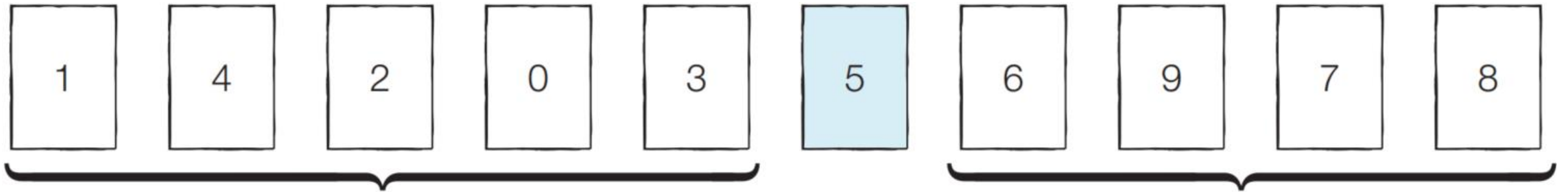
퀵 정렬 동작 예시

- [Step 2] 현재 피벗의 값은 '5'입니다. 왼쪽에서부터 '5'보다 큰 데이터를 선택하므로 '6'이 선택되고, 오른쪽에서부터 '5'보다 작은 데이터를 선택하므로 '1'이 선택됩니다. 단, 이처럼 **위치가 엇갈리는 경우 '피벗'과 '작은 데이터'의 위치를 서로 변경**합니다.



퀵 정렬 동작 예시

- [분할 완료] 이제 '5'의 왼쪽에 있는 데이터는 모두 5보다 작고, 오른쪽에 있는 데이터는 모두 '5'보다 크다는 특징이 있습니다. 이렇게 피벗을 기준으로 데이터 묶음을 나누는 작업을 분할 (Divide)이라고 합니다.



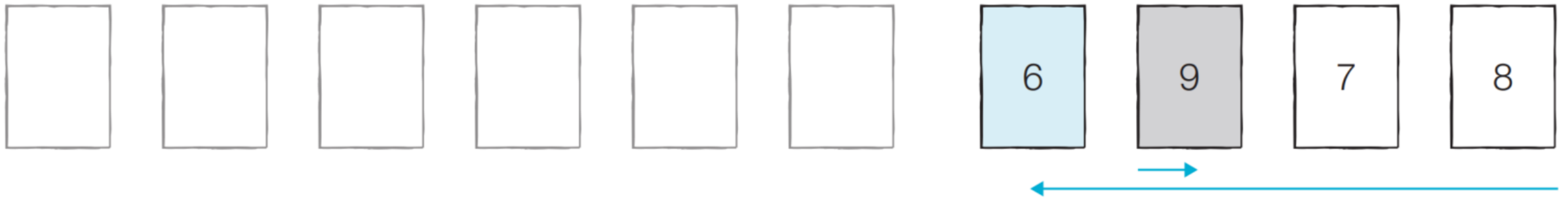
퀵 정렬 동작 예시

- [왼쪽 데이터 묶음 정렬] 왼쪽에 있는 데이터에 대해서 마찬가지로 정렬을 수행합니다.

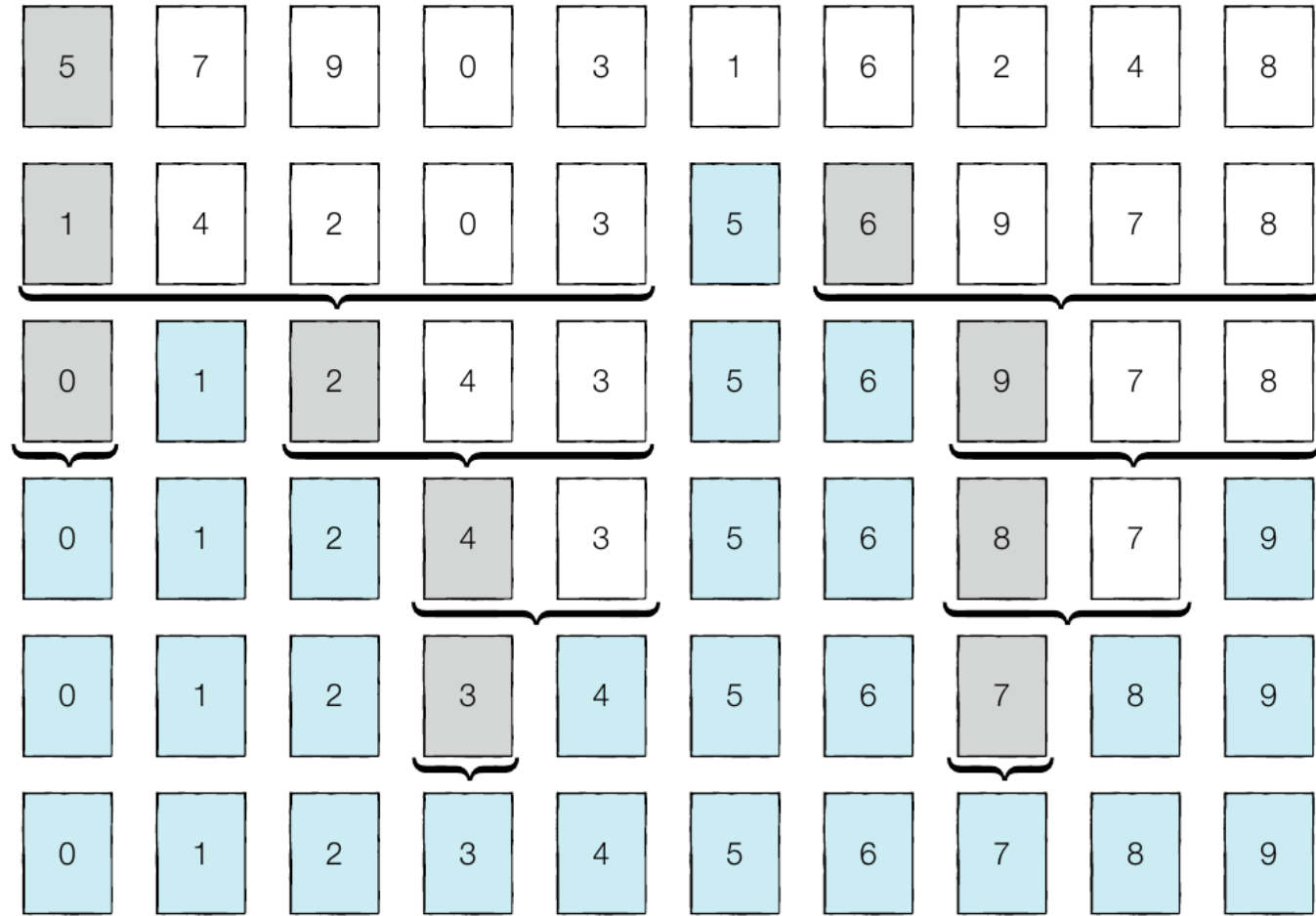


퀵 정렬 동작 예시

- [오른쪽 데이터 묶음 정렬] 오른쪽에 있는 데이터에 대해서 마찬가지로 정렬을 수행합니다.
 - 이러한 과정을 반복하면 전체 데이터에 대해서 정렬이 수행됩니다.

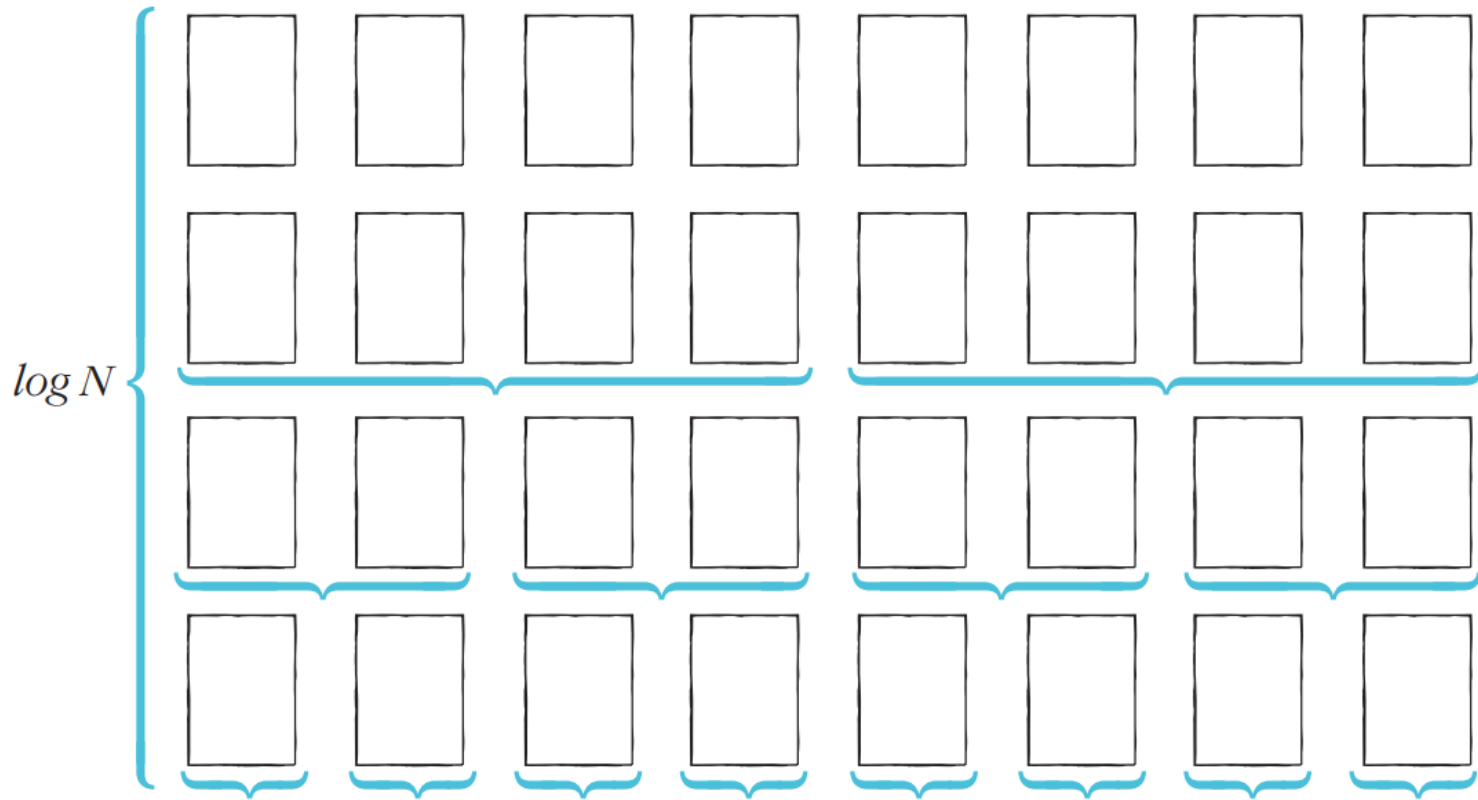


퀵 정렬 동작 예시



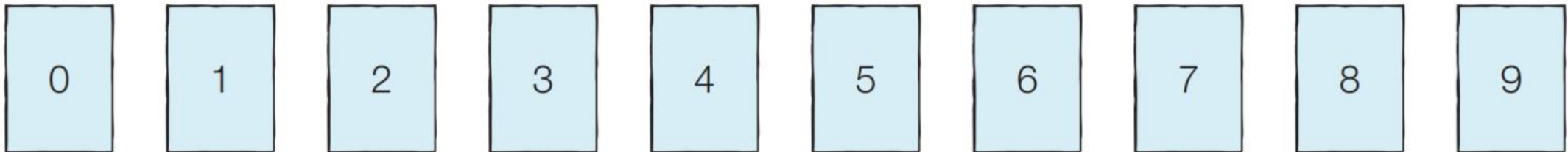
퀵 정렬이 빠른 이유: 직관적인 이해

- 이상적인 경우 분할이 절반씩 일어난다면 전체 연산 횟수로 $O(N \log N)$ 를 기대할 수 있습니다.
 - 너비 X 높이 = $N \times \log N = N \log N$



퀵 정렬의 시간 복잡도

- 퀵 정렬은 평균의 경우 $O(N \log N)$ 의 시간 복잡도를 가집니다.
- 하지만 최악의 경우 $O(N^2)$ 의 시간 복잡도를 가집니다.
 - 첫 번째 원소를 피벗으로 삼을 때, 이미 정렬된 배열에 대해서 퀵 정렬을 수행하면 어떻게 될까요?



데이터의 개수(N)	N^2 (선택 정렬, 삽입 정렬)	$N \log_2 N$ (퀵 정렬)
N = 1,000	$\approx 1,000,000$	$\approx 10,000$
N = 1,000,000	$\approx 1,000,000,000,000$	$\approx 20,000,000$

퀵 정렬 소스코드: 일반적인 방식 (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++)

```
#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

```
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]

정렬 알고리즘: 계수 정렬

계수 정렬

- 특정한 조건이 부합할 때만 사용할 수 있지만 **매우 빠르게 동작하는** 정렬 알고리즘입니다.
 - 계수 정렬은 데이터의 크기 범위가 제한되어 정수 형태로 표현할 수 있을 때 사용 가능합니다.
- 데이터의 개수가 N , 데이터(양수) 중 최대값이 K 일 때 최악의 경우에도 수행 시간 $O(N + K)$ 를 보장합니다.

계수 정렬 동작 예시

- [Step 0] 가장 작은 데이터부터 가장 큰 데이터까지의 범위가 모두 담길 수 있도록 리스트를 생성합니다.
- 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	0	0	0

계수 정렬 동작 예시

- [Step 1] 데이터를 하나씩 확인하며 데이터의 값과 동일한 인덱스의 데이터를 1씩 증가시킵니다.
- 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

계수 정렬 동작 예시

- [Step 2] 데이터를 하나씩 확인하며 데이터의 값과 동일한 인덱스의 데이터를 1씩 증가시킵니다.
- 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

계수 정렬 동작 예시

- [Step 3] 데이터를 하나씩 확인하며 데이터의 값과 동일한 인덱스의 데이터를 1씩 증가시킵니다.
- 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

계수 정렬 동작 예시

- [Step 15] 결과적으로 최종 리스트에는 각 데이터가 몇 번씩 등장했는지 그 횟수가 기록됩니다.
- 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

계수 정렬 동작 예시

- 결과를 확인할 때는 리스트의 첫 번째 데이터부터 하나씩 그 값만큼 반복하여 인덱스를 출력합니다.

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:

계수 정렬 동작 예시

- 결과를 확인할 때는 리스트의 첫 번째 데이터부터 하나씩 그 값만큼 반복하여 인덱스를 출력합니다.

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

계수 정렬 동작 예시

- 결과를 확인할 때는 리스트의 첫 번째 데이터부터 하나씩 그 값만큼 반복하여 인덱스를 출력합니다.

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

계수 정렬 동작 예시

- 결과를 확인할 때는 리스트의 첫 번째 데이터부터 하나씩 그 값만큼 반복하여 인덱스를 출력합니다.

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

계수 정렬 동작 예시

- 결과를 확인할 때는 리스트의 첫 번째 데이터부터 하나씩 그 값만큼 반복하여 인덱스를 출력합니다.

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 (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++)

```
#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 정렬 소스코드 (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 정렬의 복잡도 분석

- 계수 정렬의 시간 복잡도와 공간 복잡도는 모두 $O(N + K)$ 입니다.
- 계수 정렬은 때에 따라서 심각한 비효율성을 초래할 수 있습니다.
 - 데이터가 0과 999,999로 단 2개만 존재하는 경우를 생각해 봅시다.
- 계수 정렬은 동일한 값을 가지는 데이터가 여러 개 등장할 때 효과적으로 사용할 수 있습니다.
 - 성적의 경우 100점을 맞은 학생이 여러 명일 수 있기 때문에 계수 정렬이 효과적입니다.

정렬 알고리즘 비교

정렬 알고리즘 비교하기

- 앞서 다룬 네 가지 정렬 알고리즘을 비교하면 다음과 같습니다.
- 추가적으로 대부분의 프로그래밍 언어에서 지원하는 표준 정렬 라이브러리는 최악의 경우에도 $O(N\log N)$ 을 보장하도록 설계되어 있습니다.

정렬 알고리즘	평균 시간 복잡도	공간 복잡도	특징
선택 정렬	$O(N^2)$	$O(N)$	아이디어가 매우 간단합니다.
삽입 정렬	$O(N^2)$	$O(N)$	데이터가 거의 정렬되어 있을 때는 가장 빠릅니다.
퀵 정렬	$O(N\log N)$	$O(N)$	대부분의 경우에 가장 적합하며, 충분히 빠릅니다.
계수 정렬	$O(N + K)$	$O(N + K)$	데이터의 크기가 한정되어 있는 경우에만 사용이 가능하지만 매우 빠르게 동작합니다.

선택 정렬과 기본 정렬 라이브러리 수행 시간 비교

```
from random import randint
import time

# 배열에 10,000개의 정수를 삽입
array = []
for _ in range(10000):
    # 1부터 100 사이의 랜덤한 정수
    array.append(randint(1, 100))

# 선택 정렬 프로그램 성능 측정
start_time = time.time()

# 선택 정렬 프로그램 소스코드
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]

# 측정 종료
end_time = time.time()
# 수행 시간 출력
print("선택 정렬 성능 측정:", end_time - start_time)
```

```
# 배열을 다시 무작위 데이터로 초기화
array = []
for _ in range(10000):
    # 1부터 100 사이의 랜덤한 정수
    array.append(randint(1, 100))

# 기본 정렬 라이브러리 성능 측정
start_time = time.time()

# 기본 정렬 라이브러리 사용
array.sort()

# 측정 종료
end_time = time.time()
# 수행 시간 출력
print("기본 정렬 라이브러리 성능 측정:", end_time - start_time)
```

실행 결과

선택 정렬 성능 측정: 35.841460943222046

기본 정렬 라이브러리 성능 측정: 0.0013387203216552734

정렬 알고리즘: 문제 풀이

<문제> 두 배열의 원소 교체: 문제 설명

- 동빈이는 두 개의 배열 A와 B를 가지고 있습니다. 두 배열은 N개의 원소로 구성되어 있으며, 배열의 원소는 모두 자연수입니다.
- 동빈이는 최대 K 번의 바꿔치기 연산을 수행할 수 있는데, 바꿔치기 연산이란 배열 A에 있는 원소 하나와 배열 B에 있는 원소 하나를 골라서 두 원소를 서로 바꾸는 것을 말합니다.
- 동빈이의 최종 목표는 배열 A의 모든 원소의 합이 최대가 되도록 하는 것이며, 여러분은 동빈이를 도와야 합니다.
- N, K, 그리고 배열 A와 B의 정보가 주어졌을 때, 최대 K 번의 바꿔치기 연산을 수행하여 만들 수 있는 배열 A의 모든 원소의 합의 최댓값을 출력하는 프로그램을 작성하세요.

<문제> 두 배열의 원소 교체: 문제 설명

- 예를 들어 $N = 5$, $K = 3$ 이고, 배열 A와 B가 다음과 같다고 해봅시다.
 - 배열 A = [1, 2, 5, 4, 3]
 - 배열 B = [5, 5, 6, 6, 5]
- 이 경우, 다음과 같이 세 번의 연산을 수행할 수 있습니다.
 - 연산 1) 배열 A의 원소 '1'과 배열 B의 원소 '6'을 바꾸기
 - 연산 2) 배열 A의 원소 '2'와 배열 B의 원소 '6'을 바꾸기
 - 연산 3) 배열 A의 원소 '3'과 배열 B의 원소 '5'를 바꾸기
- 세 번의 연산 이후 배열 A와 배열 B의 상태는 다음과 같이 구성될 것입니다.
 - 배열 A = [6, 6, 5, 4, 5]
 - 배열 B = [3, 5, 1, 2, 5]
- 이때 배열 A의 모든 원소의 합은 26이 되며, 이보다 더 합을 크게 만들 수는 없습니다.

<문제> 두 배열의 원소 교체: 문제 조건

난이도 ●○○ | 풀이 시간 15분 | 시간제한 2초 | 메모리 제한 128MB

입력 조건

- 첫 번째 줄에 N, K 가 공백을 기준으로 구분되어 입력됩니다. ($1 \leq N \leq 100,000, 0 \leq K \leq N$)
- 두 번째 줄에 배열 A의 원소들이 공백을 기준으로 구분되어 입력됩니다. 모든 원소는 10,000,000보다 작은 자연수입니다.
- 세 번째 줄에 배열 B의 원소들이 공백을 기준으로 구분되어 입력됩니다. 모든 원소는 10,000,000보다 작은 자연수입니다.

출력 조건

- 최대 K번의 바꿔치기 연산을 수행하여 만들 수 있는 배열 A의 모든 원소의 합의 최댓값을 출력합니다.

입력 예시

```
5 3
1 2 5 4 3
5 5 6 6 5
```

출력 예시

26

<문제> 두 배열의 원소 교체: 문제 해결 아이디어

- 핵심 아이디어: 매번 배열 A에서 가장 작은 원소를 골라서, 배열 B에서 가장 큰 원소와 교체 합니다.
- 가장 먼저 배열 A와 B가 주어지면 A에 대하여 오름차순 정렬하고, B에 대하여 내림차순 정렬 합니다.
- 이후에 두 배열의 원소를 첫 번째 인덱스부터 차례로 확인하면서 A의 원소가 B의 원소보다 작을 때에만 교체를 수행합니다.
- 이 문제에서는 두 배열의 원소가 최대 100,000개까지 입력될 수 있으므로, 최악의 경우 $O(N\log N)$ 을 보장하는 정렬 알고리즘을 이용해야 합니다.

<문제> 두 배열의 원소 교체: 답안 예시 (Python)

```
n, k = map(int, input().split()) # N과 K를 입력 받기
a = list(map(int, input().split())) # 배열 A의 모든 원소를 입력 받기
b = list(map(int, input().split())) # 배열 B의 모든 원소를 입력 받기

a.sort() # 배열 A는 오름차순 정렬 수행
b.sort(reverse=True) # 배열 B는 내림차순 정렬 수행

# 첫 번째 인덱스부터 확인하며, 두 배열의 원소를 최대 K번 비교
for i in range(k):
    # A의 원소가 B의 원소보다 작은 경우
    if a[i] < b[i]:
        # 두 원소를 교체
        a[i], b[i] = b[i], a[i]
    else: # A의 원소가 B의 원소보다 크거나 같을 때, 반복문을 탈출
        break

print(sum(a)) # 배열 A의 모든 원소의 합을 출력
```

<문제> 두 배열의 원소 교체: 답안 예시 (C++)

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

using namespace std;

int n, k;
vector<int> a, b;

bool compare(int x, int y) {
    return x > y;
}
```

```
int main(void) {
    cin >> n >> k;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        a.push_back(x);
    }
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        b.push_back(x);
    }
    sort(a.begin(), a.end());
    sort(b.begin(), b.end(), compare);
    int result = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] < b[i]) swap(a[i], b[i]);
        else break;
    }
    for (int i = 0; i < n; i++) {
        result += a[i];
    }
    cout << result << '\n';
    return 0;
}
```

<문제> 성적이 낮은 순서로 학생 출력하기



실전 문제

성적이 낮은 순서로 학생 출력하기

난이도 ●○○ | 풀이 시간 20분 | 시간 제한 1초 | 메모리 제한 128MB | 기술 D 기업 프로그래밍 콘테스트 예선

N명의 학생 정보가 있다. 학생 정보는 학생의 이름과 학생의 성적으로 구분된다. 각 학생의 이름과 성적 정보가 주어졌을 때 성적이 낮은 순서대로 학생의 이름을 출력하는 프로그램을 작성하시오.

입력 조건

- 첫 번째 줄에 학생의 수 N이 입력된다. ($1 \leq N \leq 100,000$)
- 두 번째 줄부터 N + 1번째 줄에는 학생의 이름을 나타내는 문자열 A와 학생의 성적을 나타내는 정수 B가 공백으로 구분되어 입력된다. 문자열 A의 길이와 학생의 성적은 100 이하의 자연수이다.

출력 조건

- 모든 학생의 이름을 성적이 낮은 순서대로 출력한다. 성적이 동일한 학생들의 순서는 자유롭게 출력해도 괜찮다.

입력 예시

```
2
홍길동 95
이순신 77
```

출력 예시

```
이순신 홍길동
```

<문제> 성적이 낮은 순서로 학생 출력하기

```
# N을 입력받기
n = int(input())

# N명의 학생 정보를 입력받아 리스트에 저장
array = []
for i in range(n):
    input_data = input().split()
    # 이름은 문자열 그대로, 점수는 정수형으로 변환하여 저장
    array.append((input_data[0], int(input_data[1])))

# 키(Key)를 이용하여, 점수를 기준으로 정렬
array = sorted(array, key=lambda student: student[1])

# 정렬이 수행된 결과를 출력
for student in array:
    print(student[0], end=' ')
```

<문제> 국영수

Q

23

국영수

1회 2회 3회

☐ ☐ ☐

난이도 ●○○ | 풀이 시간 20분 | 시간 제한 1초 | 메모리 제한 256MB | 기술 핵심 유형

링크 <https://www.acmicpc.net/problem/10825>

도현이네 반 학생 N명의 이름과 국어, 영어, 수학 점수가 주어집니다. 이때, 다음과 같은 조건으로 학생의 성적을 정렬하는 프로그램을 작성하세요.

1. 국어 점수가 감소하는 순서로
2. 국어 점수가 같으면 영어 점수가 증가하는 순서로
3. 국어 점수와 영어 점수가 같으면 수학 점수가 감소하는 순서로
4. 모든 점수가 같으면 이름이 사전 순으로 증가하는 순서로 (단, 아스키코드에서 대문자는 소문자보다 작으므로 사전 순으로 앞에 옵니다.)

입력 조건

- 첫째 줄에 도현이네 반의 학생 수 N ($1 \leq N \leq 100,000$)이 주어집니다.
- 둘째 줄부터 한 줄에 하나씩 각 학생의 이름, 국어, 영어, 수학 점수가 공백으로 구분해 주어집니다.
- 점수는 1보다 크거나 같고, 100보다 작거나 같은 자연수입니다.
- 이름은 알파벳 대소문자로 이루어진 문자열이고, 길이는 10자리를 넘지 않습니다.

출력 조건

- 문제에 나와 있는 정렬 기준으로 정렬한 후 첫째 줄부터 N개의 줄에 걸쳐 각 학생의 이름을 출력합니다.

<문제> 국영수

입력 예시

```
12
Junkyu 50 60 100
Sangkeun 80 60 50
Sunyoung 80 70 100
Soong 50 60 90
Haebin 50 60 100
Kangsoo 60 80 100
Donghyuk 80 60 100
Sei 70 70 70
Wonseob 70 70 90
Sanghyun 70 70 80
nsj 80 80 80
Taewhan 50 60 90
```

출력 예시

```
Donghyuk
Sangkeun
Sunyoung
nsj
Wonseob
Sanghyun
Sei
Kangsoo
Haebin
Junkyu
Soong
Taewhan
```

<문제> 국영수

```
n = int(input())
students = [] # 학생 정보를 담을 리스트

# 모든 학생 정보를 입력받기
for _ in range(n):
    students.append(input().split())

...

[정렬 기준]
1) 두 번째 원소를 기준으로 내림차순 정렬
2) 두 번째 원소가 같은 경우, 세 번째 원소를 기준으로 오름차순 정렬
3) 세 번째 원소가 같은 경우, 네 번째 원소를 기준으로 내림차순 정렬
4) 네 번째 원소가 같은 경우, 첫 번째 원소를 기준으로 오름차순 정렬
...

students.sort(key=lambda x: (-int(x[1]), int(x[2]), -int(x[3]), x[0]))

# 정렬된 학생 정보에서 이름만 출력
for student in students:
    print(student[0])
```

실습과제

<문제> 안테나

Q

24

안테나

1회 2회 3회



난이도 ●○○ | 풀이 시간 20분 | 시간 제한 1초 | 메모리 제한 256MB | 기출 2019 SW 마에스트로 입학 테스트

링크 <https://www.acmicpc.net/problem/18310>

일직선상의 마을에 여러 채의 집이 위치해 있습니다. 이 중에서 특정 위치의 집에 특별히 한 개의 안테나를 설치하기로 결정했습니다. 효율성을 위해 안테나로부터 모든 집까지의 거리의 총합이 최소가 되도록 설치하려고 합니다. 이때 안테나는 집이 위치한 곳에만 설치할 수 있고, 논리적으로 동일한 위치에 여러 개의 집이 존재하는 것이 가능합니다.

집들의 위치 값이 주어질 때, 안테나를 설치할 위치를 선택하는 프로그램을 작성하세요.

예를 들어 $N = 4$ 이고, 각 위치가 1, 5, 7, 9일 때를 가정하겠습니다.



이 경우 5의 위치에 설치했을 때, 안테나로부터 모든 집까지의 거리의 총합이 $(4 + 0 + 2 + 4) = 10$ 으로, 최소가 됩니다.

<문제> 안테나

입력 조건

- 첫째 줄에 집의 수 N 이 자연수로 주어집니다. ($1 \leq N \leq 200,000$)
- 둘째 줄에 N 개의 집에 위치가 공백으로 구분되어 1 이상 100,000 이하의 자연수로 주어집니다.

출력 조건

- 첫째 줄에 안테나를 설치할 위치의 값을 출력합니다. 단, 안테나를 설치할 수 있는 위치 값으로 여러 개의 값이 도출될 경우 가장 작은 값을 출력합니다.

입력 예시

```
4  
5 1 7 9
```

출력 예시

```
5
```

<문제> 실패율

링크 참고: <https://school.programmers.co.kr/learn/courses/30/lessons/42889>



난이도 ●○○ | 풀이 시간 20분 | 시간 제한 1초 | 메모리 제한 128MB | 기출 2019 카카오 신입 공채 1차
링크 <https://programmers.co.kr/learn/courses/30/lessons/42889>

주의! 이 문제는 기본 코드가 제공되므로 상기 링크를 통해서 문제를 풀어야 합니다.

슈퍼 게임 개발자 오렐리는 큰 고민에 빠졌습니다. 그녀가 만든 프렌즈 오천성이 대성공을 거뒀지만, 요즘 신규 사용자의 수가 급감했습니다. 원인은 신규 사용자와 기존 사용자 사이에 스테이지 차이가 너무 큰 것이 문제였습니다.

이 문제를 어떻게 할까 고민한 그녀는 동적으로 게임 시간을 늘려서 난이도를 조절하기로 했습니다. 역시 슈퍼 개발자라 대부분의 로직은 쉽게 구현했지만, 실패율을 구하는 부분에서 위기에 빠지고 말았습니다. 오렐리를 위해 실패율을 구하는 코드를 완성하세요.

실패율은 다음과 같이 정의합니다.

- 스테이지에 도달했으나 아직 클리어하지 못한 플레이어의 수 / 스테이지에 도달한 플레이어의 수

전체 스테이지의 개수 N, 게임을 이용하는 사용자가 현재 멈춰있는 스테이지의 번호가 담긴 배열 stages가 매개변수로 주어질 때, 실패율이 높은 스테이지부터 내림차순으로 스테이지의 번호가 담겨 있는 배열을 return 하도록 solution 함수를 완성하세요.

이진 탐색 알고리즘

이진 탐색 알고리즘

- 순차 탐색: 리스트 안에 있는 특정한 데이터를 찾기 위해 앞에서부터 데이터를 하나씩 확인하는 방법
- 이진 탐색: 정렬되어 있는 리스트에서 탐색 범위를 절반씩 좁혀가며 데이터를 탐색하는 방법
 - 이진 탐색은 시작점, 끝점, 중간점을 이용하여 탐색 범위를 설정합니다.

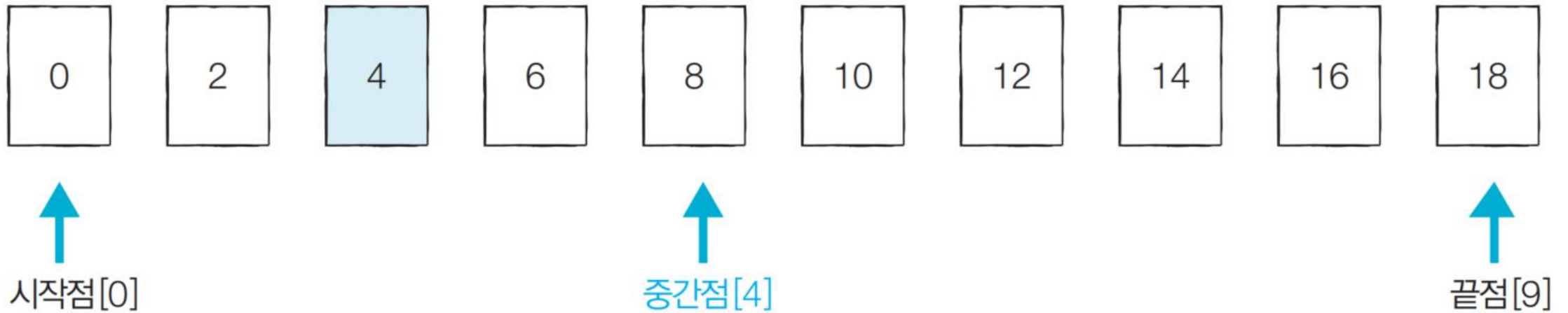
이진 탐색 동작 예시

- 이미 정렬된 10개의 데이터 중에서 값이 4인 원소를 찾는 예시를 살펴봅시다.



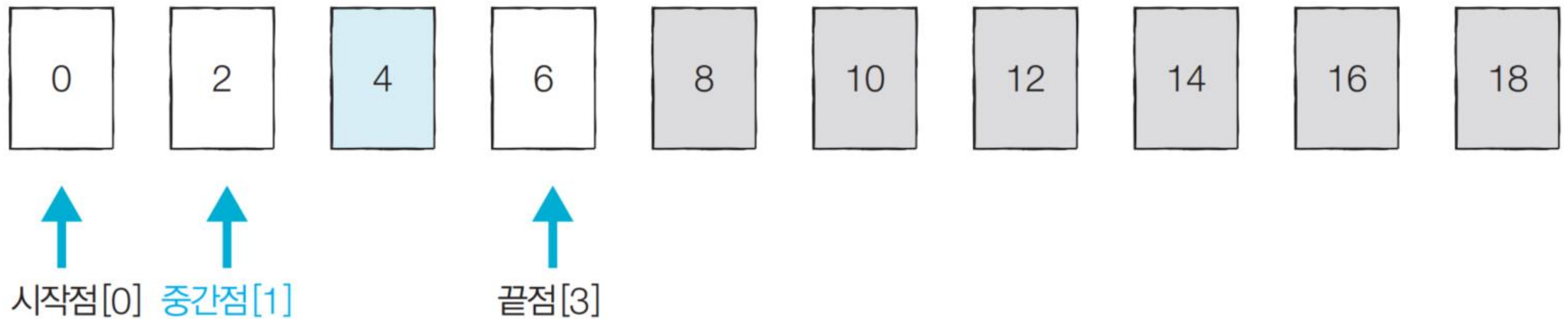
이진 탐색 동작 예시

- [Step 1] 시작점: 0, 끝점: 9, 중간점: 4 (소수점 이하 제거)



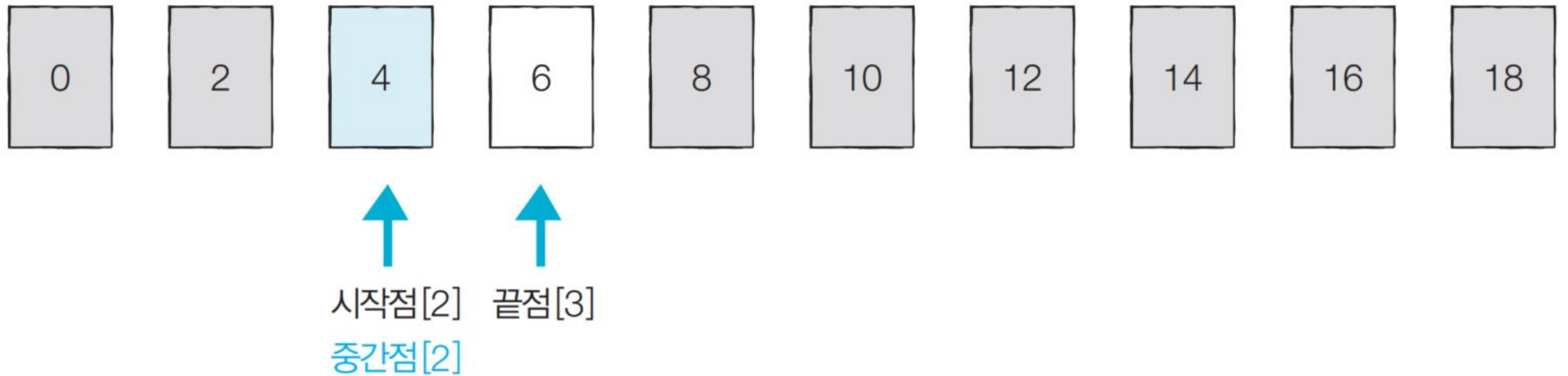
이진 탐색 동작 예시

- [Step 2] 시작점: 0, 끝점: 3, 중간점: 1 (소수점 이하 제거)



이진 탐색 동작 예시

- [Step 3] 시작점: 2, 끝점: 3, 중간점: 2 (소수점 이하 제거)



이진 탐색의 시간 복잡도

- 단계마다 탐색 범위를 2로 나누는 것과 동일하므로 연산 횟수는 $\log_2 N$ 에 비례합니다.
- 예를 들어 초기 데이터 개수가 32개일 때, 이상적으로 1단계를 거치면 16개가량의 데이터만 남습니다.
 - 2단계를 거치면 8개가량의 데이터만 남습니다.
 - 3단계를 거치면 4개가량의 데이터만 남습니다.
- 다시 말해 이진 탐색은 탐색 범위를 절반씩 줄이며, 시간 복잡도는 $O(\log N)$ 을 보장합니다.

이진 탐색 소스코드: 재귀적 구현 (Python)

```
# 이진 탐색 소스코드 구현 (재귀 함수)
def binary_search(array, target, start, end):
    if start > end:
        return None
    mid = (start + end) // 2
    # 찾은 경우 중간점 인덱스 반환
    if array[mid] == target:
        return mid
    # 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
    elif array[mid] > target:
        return binary_search(array, target, start, mid - 1)
    # 중간점의 값보다 찾고자 하는 값이 큰 경우 오른쪽 확인
    else:
        return binary_search(array, target, mid + 1, end)

# n(원소의 개수)과 target(찾고자 하는 값)을 입력 받기
n, target = list(map(int, input().split()))
# 전체 원소 입력 받기
array = list(map(int, input().split()))

# 이진 탐색 수행 결과 출력
result = binary_search(array, target, 0, n - 1)
if result == None:
    print("원소가 존재하지 않습니다.")
else:
    print(result + 1)
```

```
10 7 ↵
1 3 5 7 9 11 13 15 17 19 ↵
4
```

```
10 7 ↵
1 3 5 6 9 11 13 15 17 19 ↵
원소가 존재하지 않습니다.
```

이진 탐색 소스코드: 반복문 구현 (Python)

```
# 이진 탐색 소스코드 구현 (반복문)
def binary_search(array, target, start, end):
    while start <= end:
        mid = (start + end) // 2
        # 찾은 경우 중간점 인덱스 반환
        if array[mid] == target:
            return mid
        # 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
        elif array[mid] > target:
            end = mid - 1
        # 중간점의 값보다 찾고자 하는 값이 큰 경우 오른쪽 확인
        else:
            start = mid + 1
    return None

# n(원소의 개수)과 target(찾고자 하는 값)을 입력 받기
n, target = list(map(int, input().split()))
# 전체 원소 입력 받기
array = list(map(int, input().split()))

# 이진 탐색 수행 결과 출력
result = binary_search(array, target, 0, n - 1)
if result == None:
    print("원소가 존재하지 않습니다.")
else:
    print(result + 1)
```

```
10 7 ↵
1 3 5 7 9 11 13 15 17 19 ↵
4
```

```
10 7 ↵
1 3 5 6 9 11 13 15 17 19 ↵
원소가 존재하지 않습니다.
```

이진 탐색 소스코드: 반복문 구현 (C++)

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

using namespace std;

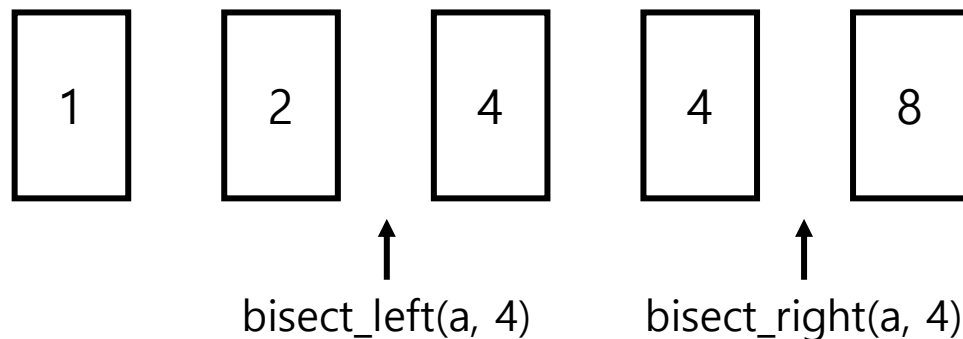
int binarySearch(vector<int>& arr, int target, int start, int end) {
    while (start <= end) {
        int mid = (start + end) / 2;
        // 찾은 경우 중간점 인덱스 반환
        if (arr[mid] == target) return mid;
        // 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
        else if (arr[mid] > target) end = mid - 1;
        // 중간점의 값보다 찾고자 하는 값이 큰 경우 오른쪽 확인
        else start = mid + 1;
    }
    return -1;
}
```

```
int n, target;
vector<int> arr;

int main(void) {
    // n(원소의 개수)와 target(찾고자 하는 값)을 입력 받기
    cin >> n >> target;
    // 전체 원소 입력 받기
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        arr.push_back(x);
    }
    // 이진 탐색 수행 결과 출력
    int result = binarySearch(arr, target, 0, n - 1);
    if (result == -1) {
        cout << "원소가 존재하지 않습니다." << '\n';
    }
    else {
        cout << result + 1 << '\n';
    }
    return 0;
}
```

파이썬 이진 탐색 라이브러리

- `bisect_left(a, x)`: 정렬된 순서를 유지하면서 배열 `a`에 `x`를 삽입할 가장 왼쪽 인덱스를 반환
- `bisect_right(a, x)`: 정렬된 순서를 유지하면서 배열 `a`에 `x`를 삽입할 가장 오른쪽 인덱스를 반환



```
from bisect import bisect_left, bisect_right
```

```
a = [1, 2, 4, 4, 8]  
x = 4
```

```
print(bisect_left(a, x))  
print(bisect_right(a, x))
```

실행 결과

2
4

값이 특정 범위에 속하는 데이터 개수 구하기

```
from bisect import bisect_left, bisect_right

# 값이 [left_value, right_value]인 데이터의 개수를 반환하는 함수
def count_by_range(a, left_value, right_value):
    right_index = bisect_right(a, right_value)
    left_index = bisect_left(a, left_value)
    return right_index - left_index

# 배열 선언
a = [1, 2, 3, 3, 3, 3, 4, 4, 8, 9]

# 값이 4인 데이터 개수 출력
print(count_by_range(a, 4, 4))

# 값이 [-1, 3] 범위에 있는 데이터 개수 출력
print(count_by_range(a, -1, 3))
```

실행 결과

2
6

파라메트릭 서치 (Parametric Search)

- 파라메트릭 서치란 최적화 문제를 결정 문제('예' 혹은 '아니오')로 바꾸어 해결하는 기법입니다.
 - 예시: 특정한 조건을 만족하는 가장 알맞은 값을 빠르게 찾는 최적화 문제
- 일반적으로 코딩 테스트에서 파라메트릭 서치 문제는 이진 탐색을 이용하여 해결할 수 있습니다.

<문제> 떡볶이 떡 만들기: 문제 설명

- 오늘 동빈이는 여행 가신 부모님을 대신해서 떡집 일을 하기로 했습니다. 오늘은 떡볶이 떡을 만드는 날입니다. 동빈이네 떡볶이 떡은 재밋게도 떡볶이 떡의 길이가 일정하지 않습니다. 대신에 한 봉지 안에 들어가는 떡의 총 길이는 절단기로 잘라서 맞춰줍니다.
- 절단기에 높이(H)를 지정하면 줄지어진 떡을 한 번에 절단합니다. 높이가 H 보다 긴 떡은 H 위의 부분이 잘릴 것이고, 낮은 떡은 잘리지 않습니다.
- 예를 들어 높이가 19, 14, 10, 17cm인 떡이 나란히 있고 절단기 높이를 15cm로 지정하면 자른 뒤 떡의 높이는 15, 14, 10, 15cm가 될 것입니다. 잘린 떡의 길이는 차례대로 4, 0, 0, 2cm입니다. 손님은 6cm만큼의 길이를 가져갑니다.
- 손님이 왔을 때 요청한 총 길이가 M 일 때 적어도 M 만큼의 떡을 얻기 위해 절단기에 설정할 수 있는 높이의 최댓값을 구하는 프로그램을 작성하세요.

<문제> 떡볶이 떡 만들기: 문제 조건

난이도 ●●○ | 풀이 시간 40분 | 시간제한 2초 | 메모리 제한 128MB

입력 조건

- 첫째 줄에 떡의 개수 N 과 요청한 떡의 길이 M 이 주어집니다. ($1 \leq N \leq 1,000,000$, $1 \leq M \leq 2,000,000,000$)
- 둘째 줄에는 떡의 개별 높이가 주어집니다. 떡 높이의 총합은 항상 M 이상이므로, 손님은 필요한 양만큼 떡을 사갈 수 있습니다. 높이는 10억보다 작거나 같은 양의 정수 또는 0입니다.

출력 조건

- 적어도 M 만큼의 떡을 집에 가져가기 위해 절단기에 설정할 수 있는 높이의 최댓값을 출력합니다.

입력 예시

```
4 6
19 15 10 17
```

출력 예시

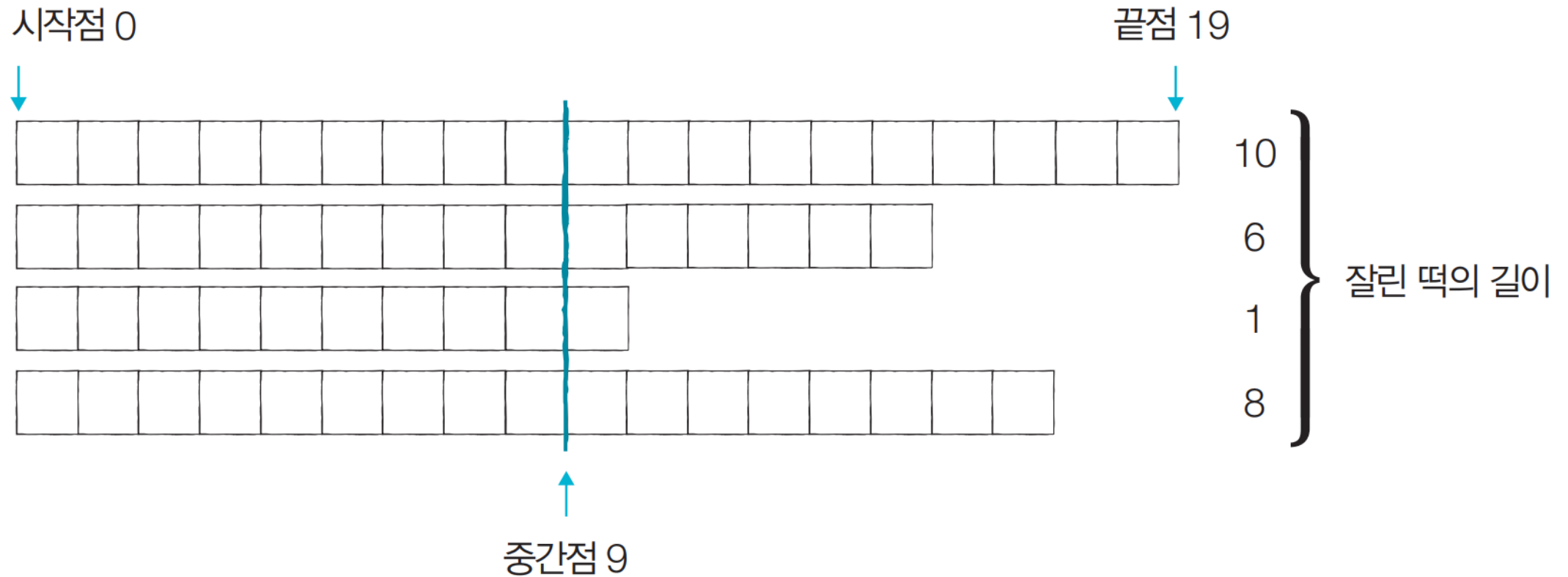
```
15
```

<문제> 떡볶이 떡 만들기: 문제 해결 아이디어

- 적절한 높이를 찾을 때까지 이진 탐색을 수행하여 높이 H 를 반복해서 조정하면 됩니다.
- '현재 이 높이로 자르면 조건을 만족할 수 있는가?'를 확인한 뒤에 조건의 만족 여부('예' 혹은 '아니오')에 따라서 탐색 범위를 좁혀서 해결할 수 있습니다.
- 절단기의 높이는 0부터 10억까지의 정수 중 하나입니다.
 - 이렇게 큰 탐색 범위를 보면 가장 먼저 이진 탐색을 떠올려야 합니다.
- 문제에서 제시된 그림 예시를 통해 이해해 봅시다.

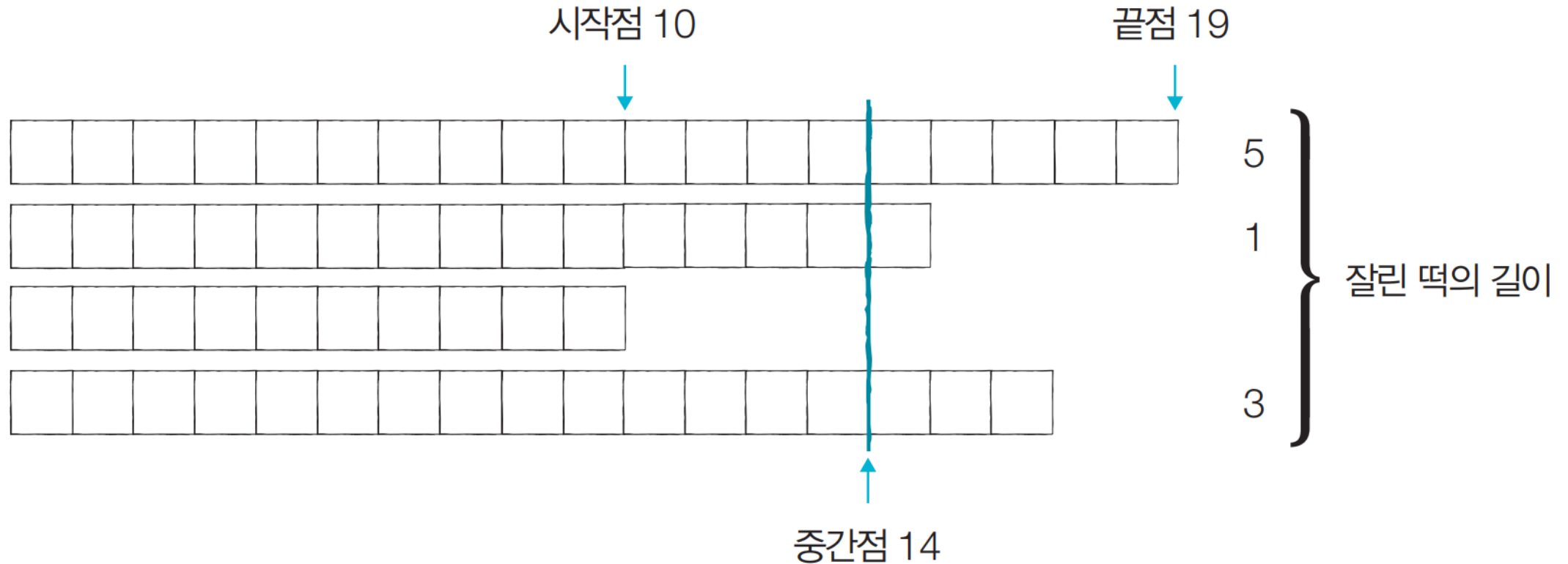
<문제> 떡볶이 떡 만들기: 문제 해결 아이디어

- [Step 1] 시작점: 0, 끝점: 19, 중간점: 9 이때 필요한 떡의 크기: $M = 6$ 이므로, 결과 저장



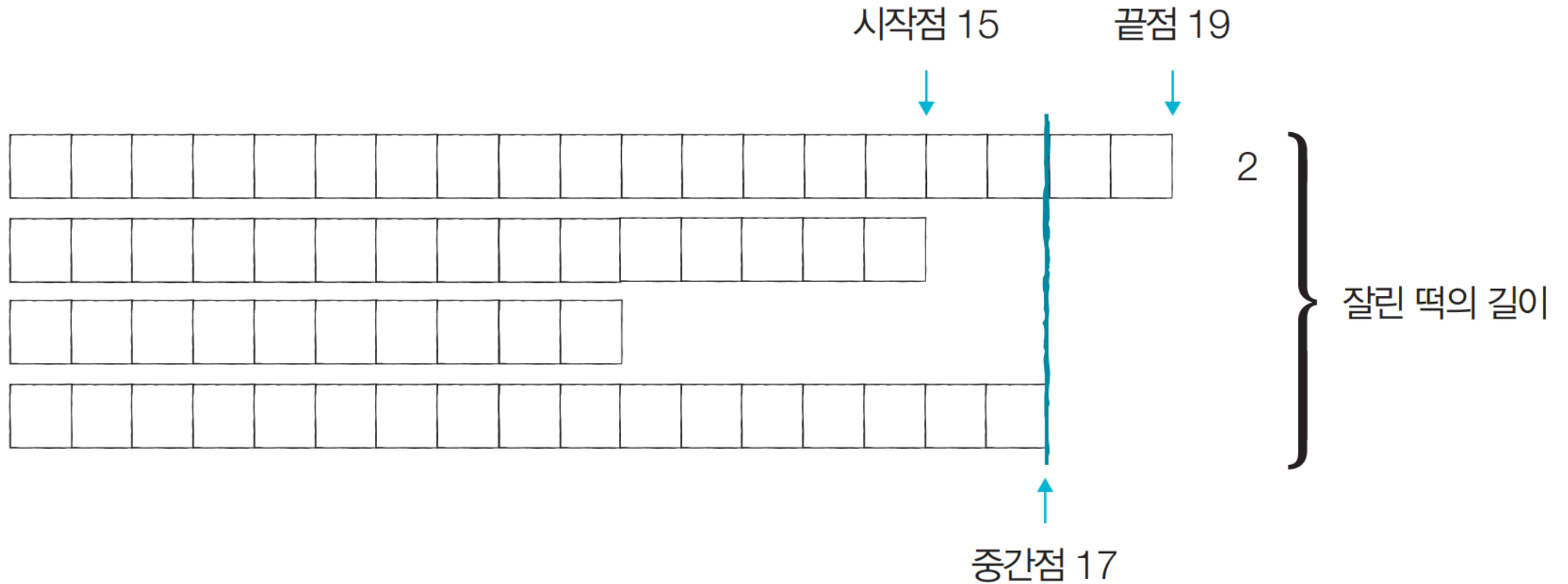
<문제> 떡볶이 떡 만들기: 문제 해결 아이디어

- [Step 2] 시작점: 10, 끝점: 19, 중간점: 14 이때 필요한 떡의 크기: $M = 6$ 이므로, 결과 저장



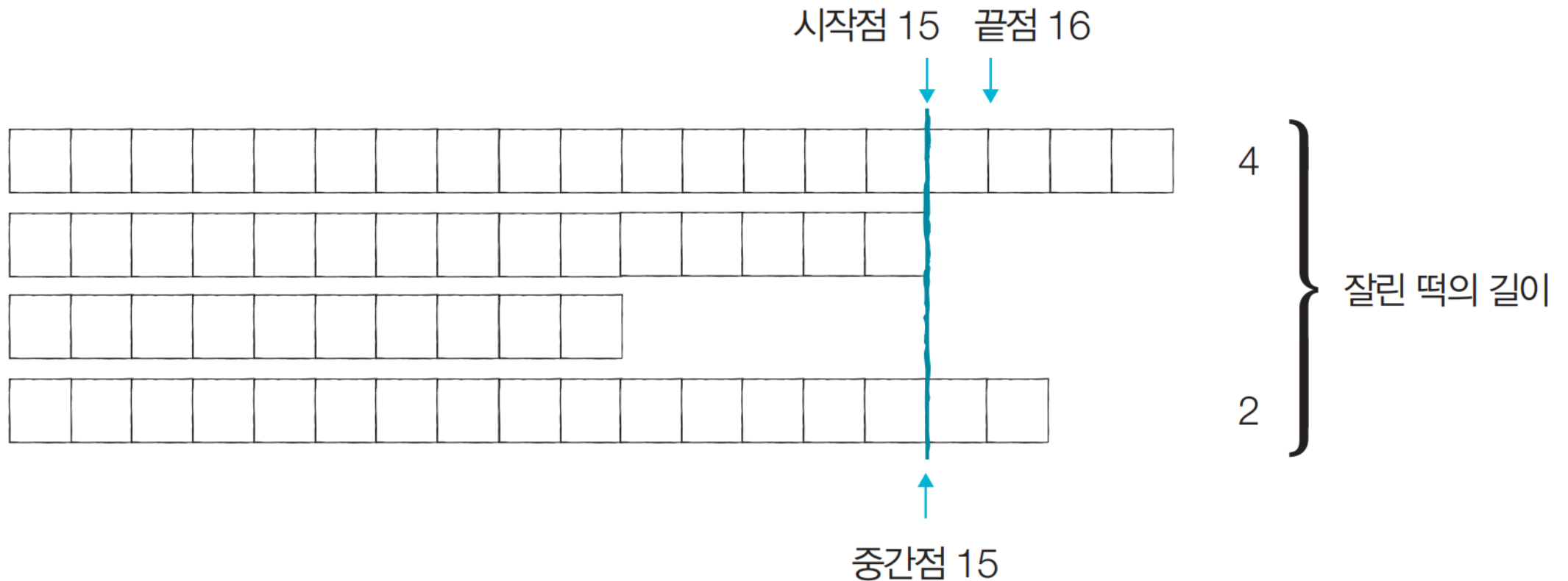
<문제> 떡볶이 떡 만들기: 문제 해결 아이디어

- [Step 3] 시작점: 15, 끝점: 19, 중간점: 17 이때 필요한 떡의 크기: $M = 6$ 이므로, 결과 저장하지 않음



<문제> 떡볶이 떡 만들기: 문제 해결 아이디어

- [Step 4] 시작점: 15, 끝점: 16, 중간점: 15 이때 필요한 떡의 크기: $M = 6$ 이므로, 결과 저장



<문제> 떡볶이 떡 만들기: 문제 해결 아이디어

- 이러한 이진 탐색 과정을 반복하면 답을 도출할 수 있습니다.
- 중간점의 값은 시간이 지날수록 '최적화된 값'이 되기 때문에, 과정을 반복하면서 얻을 수 있는 떡의 길이 합이 필요한 떡의 길이보다 크거나 같을 때마다 중간점의 값을 기록하면 됩니다.

<문제> 떡볶이 떡 만들기: 답안 예시 (Python)

```
# 떡의 개수(N)와 요청한 떡의 길이(M)을 입력
n, m = list(map(int, input().split(' ')))
# 각 떡의 개별 높이 정보를 입력
array = list(map(int, input().split()))

# 이진 탐색을 위한 시작점과 끝점 설정
start = 0
end = max(array)

# 이진 탐색 수행 (반복적)
result = 0
while(start <= end):
    total = 0
    mid = (start + end) // 2
    for x in array:
        # 잘랐을 때의 떡의 양 계산
        if x > mid:
            total += x - mid
    # 떡의 양이 부족한 경우 더 많이 자르기 (왼쪽 부분 탐색)
    if total < m:
        end = mid - 1
    # 떡의 양이 충분한 경우 덜 자르기 (오른쪽 부분 탐색)
    else:
        result = mid # 최대한 덜 잘랐을 때가 정답이므로, 여기에서 result에 기록
        start = mid + 1

# 정답 출력
print(result)
```

<문제> 떡볶이 떡 만들기: 답안 예시 (C++)

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

using namespace std;

// 떡의 개수(N)와 요청한 떡의 길이(M)
int n, m;
// 각 떡의 개별 높이 정보
vector<int> arr;
```

```
int main(void) {
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        arr.push_back(x);
    }
    // 이진 탐색을 위한 시작점과 끝점 설정
    int start = 0;
    int end = 1e9;
    // 이진 탐색 수행 (반복적)
    int result = 0;
    while (start <= end) {
        long long int total = 0;
        int mid = (start + end) / 2;
        for (int i = 0; i < n; i++) {
            // 잘랐을 때의 떡의 양 계산
            if (arr[i] > mid) total += arr[i] - mid;
        }
        if (total < m) { // 떡의 양이 부족한 경우 더 많이 자르기 (왼쪽 부분 탐색)
            end = mid - 1;
        }
        else { // 떡의 양이 충분한 경우 덜 자르기 (오른쪽 부분 탐색)
            result = mid; // 최대한 덜 잘랐을 때가 정답이므로, 여기에서 result에 기록
            start = mid + 1;
        }
    }
    cout << result << '\n';
}
```

<문제> 정렬된 배열에서 특정 수의 개수 구하기: 문제 설명

- N개의 원소를 포함하고 있는 수열이 오름차순으로 정렬되어 있습니다. 이때 이 수열에서 x가 등장하는 횟수를 계산하세요. 예를 들어 수열 {1, 1, 2, 2, 2, 2, 3}이 있을 때 $x = 2$ 라면, 현재 수열에서 값이 2인 원소가 4개이므로 4를 출력합니다.
- 단, 이 문제는 시간 복잡도 $O(\log N)$ 으로 알고리즘을 설계하지 않으면 시간 초과 판정을 받습니다.

<문제> 정렬된 배열에서 특정 수의 개수 구하기: 문제 조건

난이도 ●●○ | 풀이 시간 30분 | 시간 제한 1초 | 메모리 제한 128MB | 기출 Zoho 인터뷰

입력 조건

- 첫째 줄에 N 과 x 가 정수 형태로 공백으로 구분되어 입력됩니다.
($1 \leq N \leq 1,000,000$), ($-10^9 \leq x \leq 10^9$)
- 둘째 줄에 N 개의 원소가 정수 형태로 공백으로 구분되어 입력됩니다.
($-10^9 \leq \text{각 원소의 값} \leq 10^9$)

출력 조건

- 수열의 원소 중에서 값이 x 인 원소의 개수를 출력합니다. 단, 값이 x 인 원소가 하나도 없다면 -1 을 출력합니다.

입력 예시 1

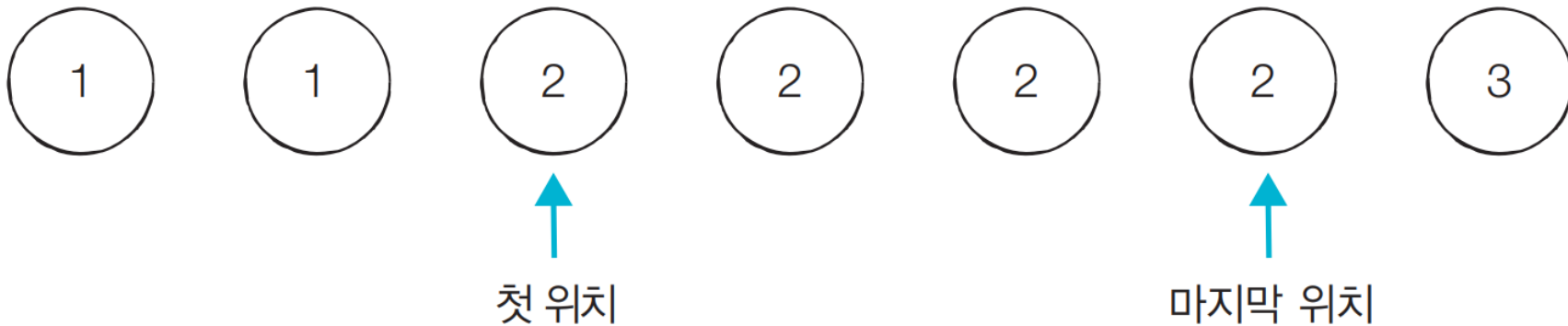
```
7 2
1 1 2 2 2 2 3
```

출력 예시 1

```
4
```

<문제> 정렬된 배열에서 특정 수의 개수 구하기: 문제 해결 아이디어

- 시간 복잡도 $O(\log N)$ 으로 동작하는 알고리즘을 요구하고 있습니다.
 - 일반적인 선형 탐색(Linear Search)로는 시간 초과 판정을 받습니다.
 - 하지만 데이터가 정렬되어 있기 때문에 **이진 탐색을 수행**할 수 있습니다.
- 특정 값이 등장하는 첫 번째 위치와 마지막 위치를 찾아 위치 차이를 계산해 문제를 해결할 수 있습니다.



<문제> 정렬된 배열에서 특정 수의 개수 구하기: 답안 예시 (Python)

```
from bisect import bisect_left, bisect_right

# 값이 [left_value, right_value]인 데이터의 개수를 반환하는 함수
def count_by_range(array, left_value, right_value):
    right_index = bisect_right(array, right_value)
    left_index = bisect_left(array, left_value)
    return right_index - left_index

n, x = map(int, input().split()) # 데이터의 개수 N, 찾고자 하는 값 x 입력받기
array = list(map(int, input().split())) # 전체 데이터 입력받기

# 값이 [x, x] 범위에 있는 데이터의 개수 계산
count = count_by_range(array, x, x)

# 값이 x인 원소가 존재하지 않는다면
if count == 0:
    print(-1)
# 값이 x인 원소가 존재한다면
else:
    print(count)
```