실습
(주사위 합의 모든 경로)

```
def test_find_dice_probabilities():
    n_faces = 6
    S = 5
    results = find_dice_probabilities(S, n_faces)
    print(results)
    assert(results[0] == len(results[1]))
    print("테스트 통과!")


test_find_dice_probabilities()
```

이진 탐색

이진 탐색

- 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
- 시작점, 끝점, 중간점을 이용하여 탐색 범위를 설정

이진 탐색 (Binary Search)

정렬된 원소 리스트를 입력받고, 원하는 원소가 있으면 그 원소의 위치를, 아니면 null 반환

탐색 범위의 중간부터 시작하여 비교를 통해 절반의 숫자 제거

매 단계마다 남은 범위의 가운데 수를 말하고 대답에 따라 절반 제거 반복

Ex) 1~100 중 하나의 숫자를 추측하여 맞추는 경우

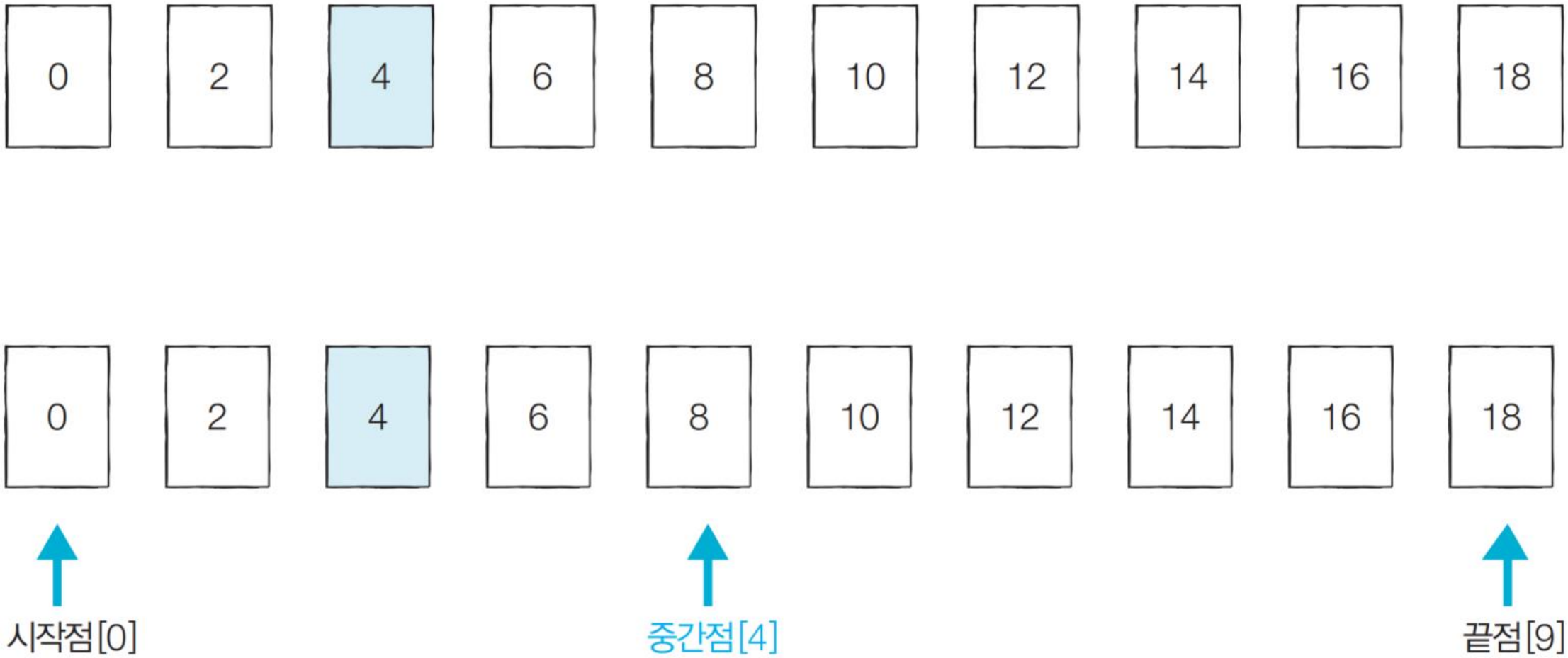
1부터 시작하여 하나씩 검토하는 단순 탐색은 비효율적

50부터 시작하여 단계마다 절반의 숫자 제거

최대 7번 만에 정답에 도달

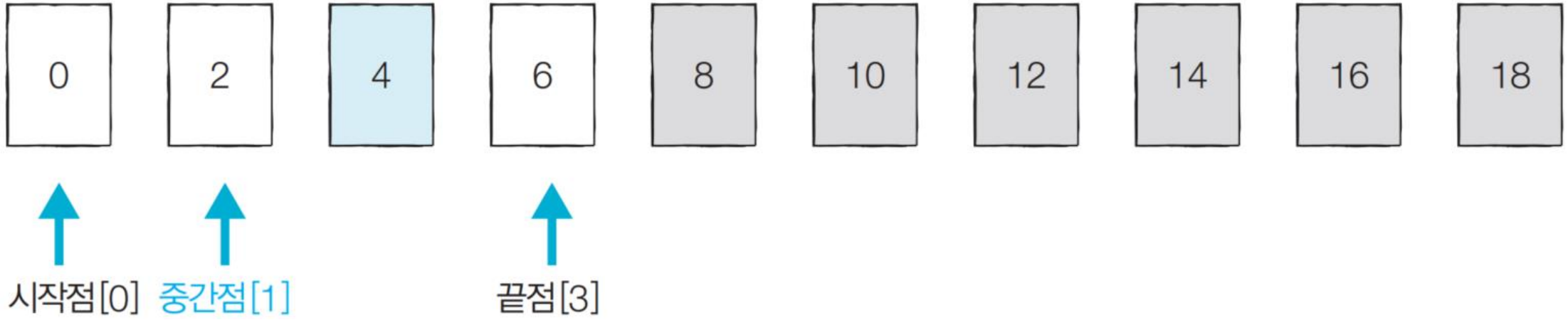
이진 탐색 동작 예시

- 이미 정렬된 10개의 데이터 중에서 값이 4인 원소를 찾는 예시

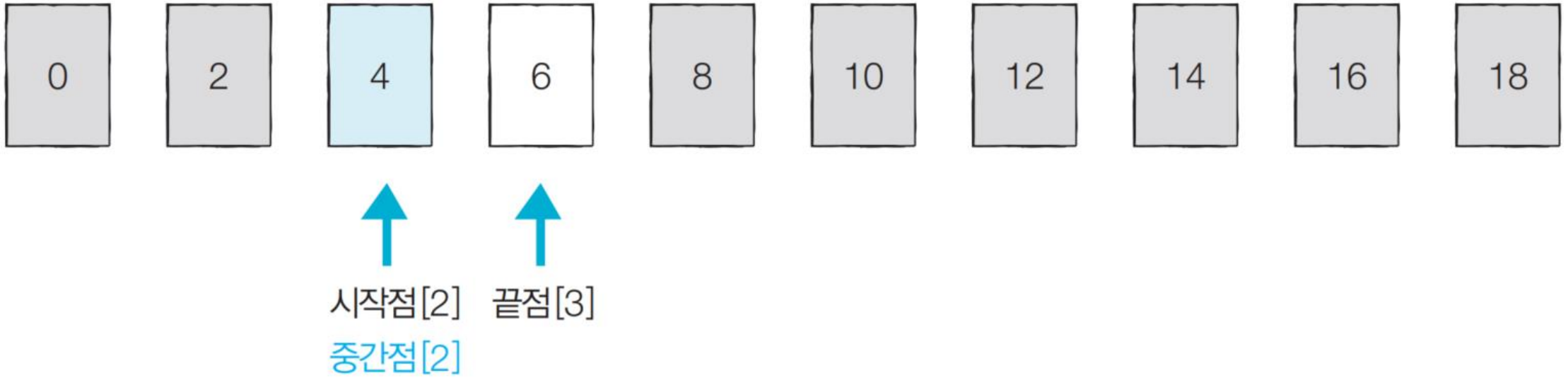


이진 탐색 동작 예시

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



- [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 비례
- 예를 들어 초기 데이터 개수가 32개일 때, 이상적으로 1단계를 거치면 16개가량의 데이터만 남는다.
  - 2단계를 거치면 8개가량의 데이터만 남는다.
  - 3단계를 거치면 4개가량의 데이터만 남는다.
- 다시 말해 이진 탐색은 탐색 범위를 절반씩 줄이며, 시간 복잡도는 $\boldsymbol{O(\log N)}$을 보장

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

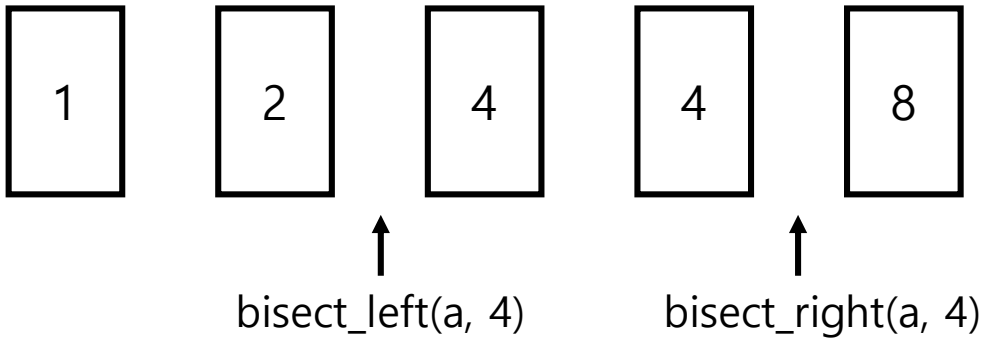
```python
# 이진 탐색 소스코드 구현 (반복문)
def binary_search(array, target, start, end):
    while start <= end:
        mid = (start + end) // 2
        # 찾은 경우 중간점 인덱스 반환
        if array[mid] == target:
            return mid
        # 중간점의 값보다 찾고자 하는 값이 작은 경우 왼쪽 확인
        elif array[mid] > target:
            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.
```

파이썬 이진 탐색 라이브러리

- bisect_left(a, x): 정렬된 순서를 유지하면서 배열 a에 x를 삽입할 가장 왼쪽 인덱스를 반환
- bisect_right(a, x): 정렬된 순서를 유지하면서 배열 a에 x를 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

```python
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
```

<문제> 떡볶이 떡 만들기: 문제 설명

- 오늘 동빈이는 여행 가신 부모님을 대신해서 떡집 일을 하기로 했다. 오늘은 떡볶이 떡을 만드는 날이다. 동빈이네 떡볶이 떡은 재밌게도 떡볶이 떡의 길이가 일정하지 않다. 대신에 한 봉지 안에 들어가는 떡의 총 길이는 절단기로 잘라서 맞춰준다.
- 절단기에 높이(H)를 지정하면 줄지어진 떡을 한 번에 절단한다. 높이가 H보다 긴 떡은 H 위의 부분이 잘릴 것이고, 낮은 떡은 잘리지 않는다.
- 예를 들어 높이가 19, 14, 10, 17cm인 떡이 나란히 있고 절단기 높이를 15cm로 지정하면 자른 뒤 떡의 높이는 15, 14, 10, 15cm가 될 것이다. 잘린 떡의 길이는 차례대로 4, 0, 0, 2cm입니다. 손님은 6cm만큼의 길이를 가져간다.
- 손님이 왔을 때 요청한 총 길이가 M일 때 적어도 M만큼의 떡을 얻기 위해 절단기에 설정할 수 있는 높이의 최댓값을 구하는 프로그램을 작성하라.

입력 조건

- 첫째 줄에 떡의 개수 N과 요청한 떡의 길이 M이 주어집니다. (1 ≤ N ≤ 1,000,000, 1 ≤ M ≤ 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 M만큼의 떡을 집에 가져가기 위해 절단기에 설정할 수 있는 높이의 최댓값을 출력합니다.

입력 예시

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

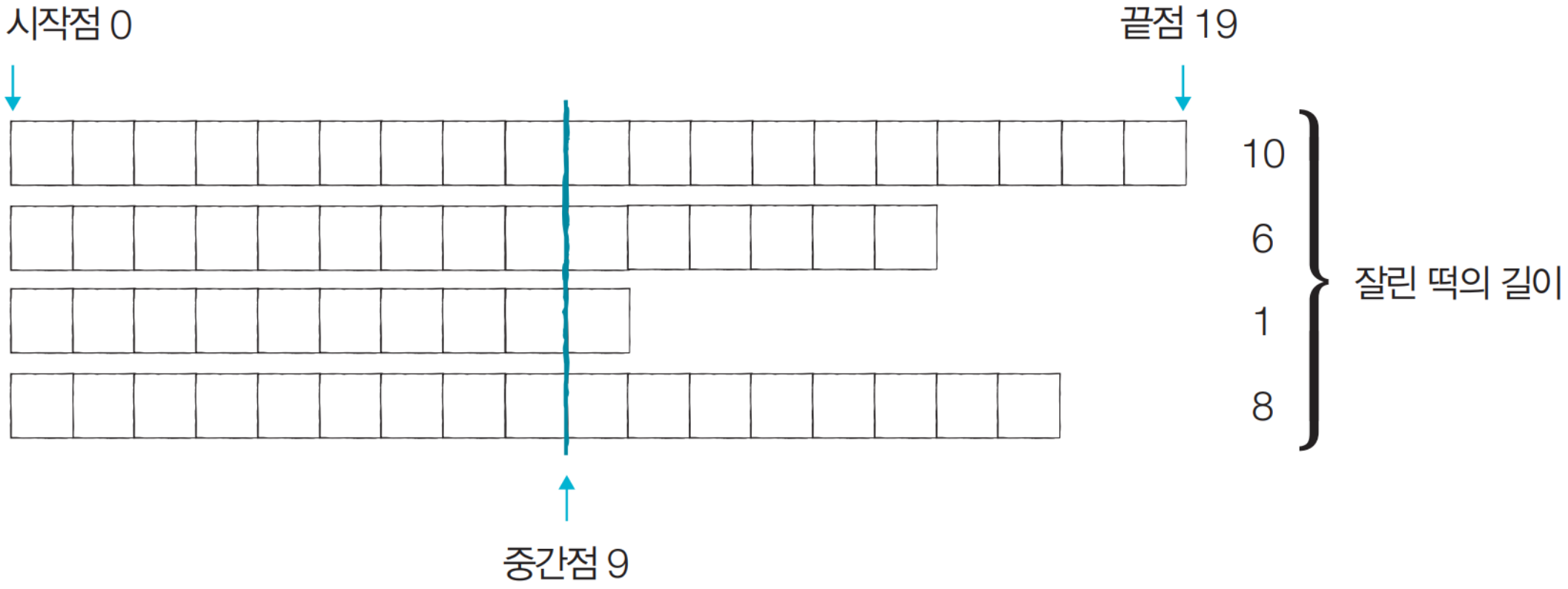
출력 예시

```
15
```

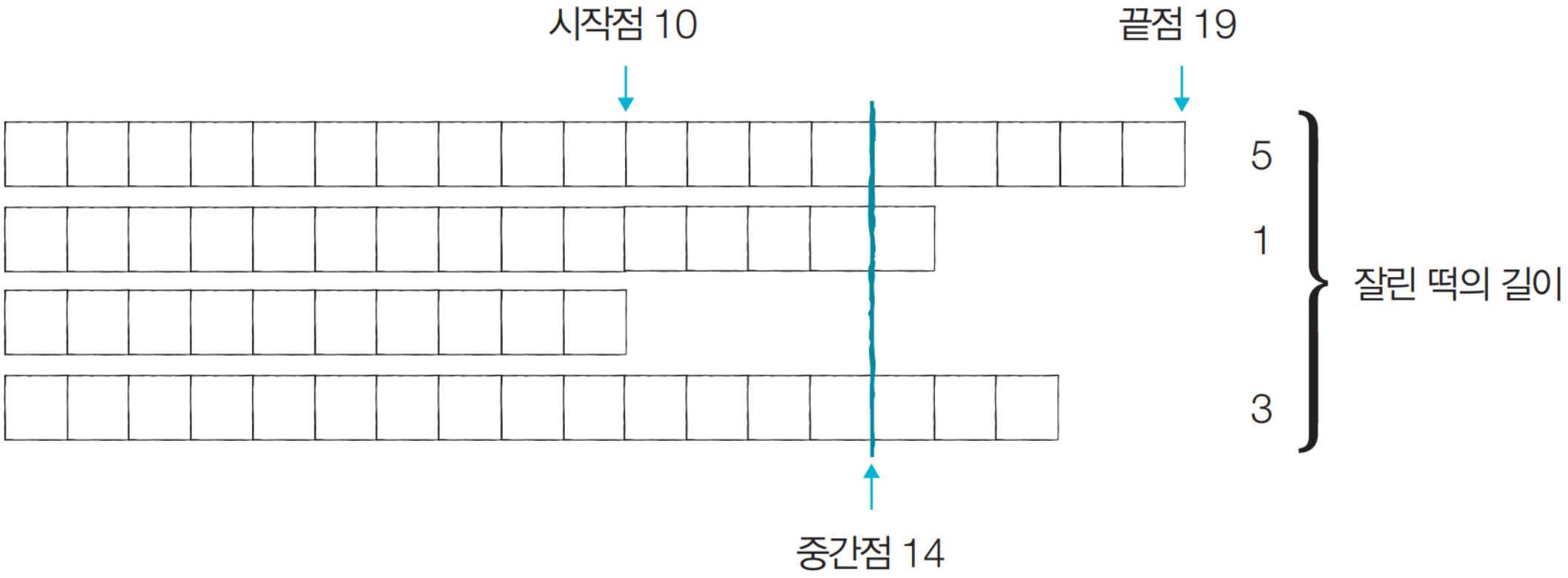
아이디어

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

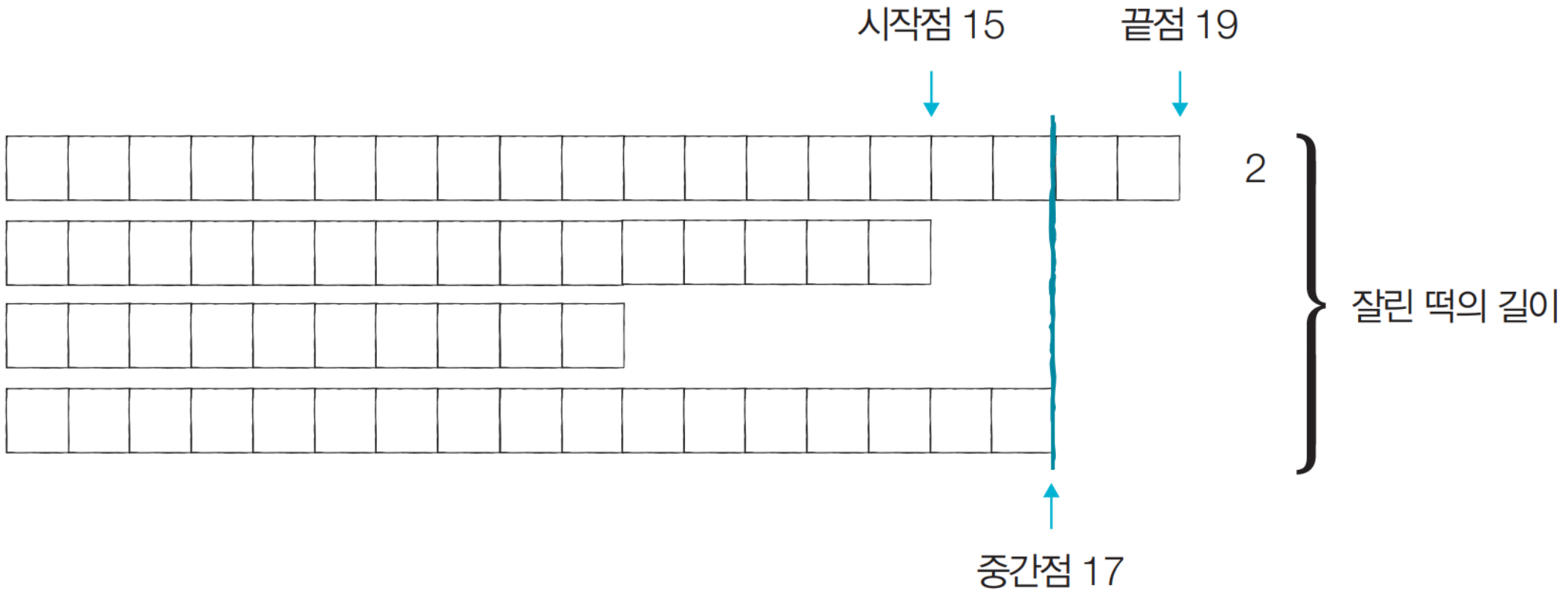
- [Step 1] 시작점: 0, 끝점: 19, 중간점: 9　　이때 필요한 떡의 크기: M = 6이므로, 결과 저장



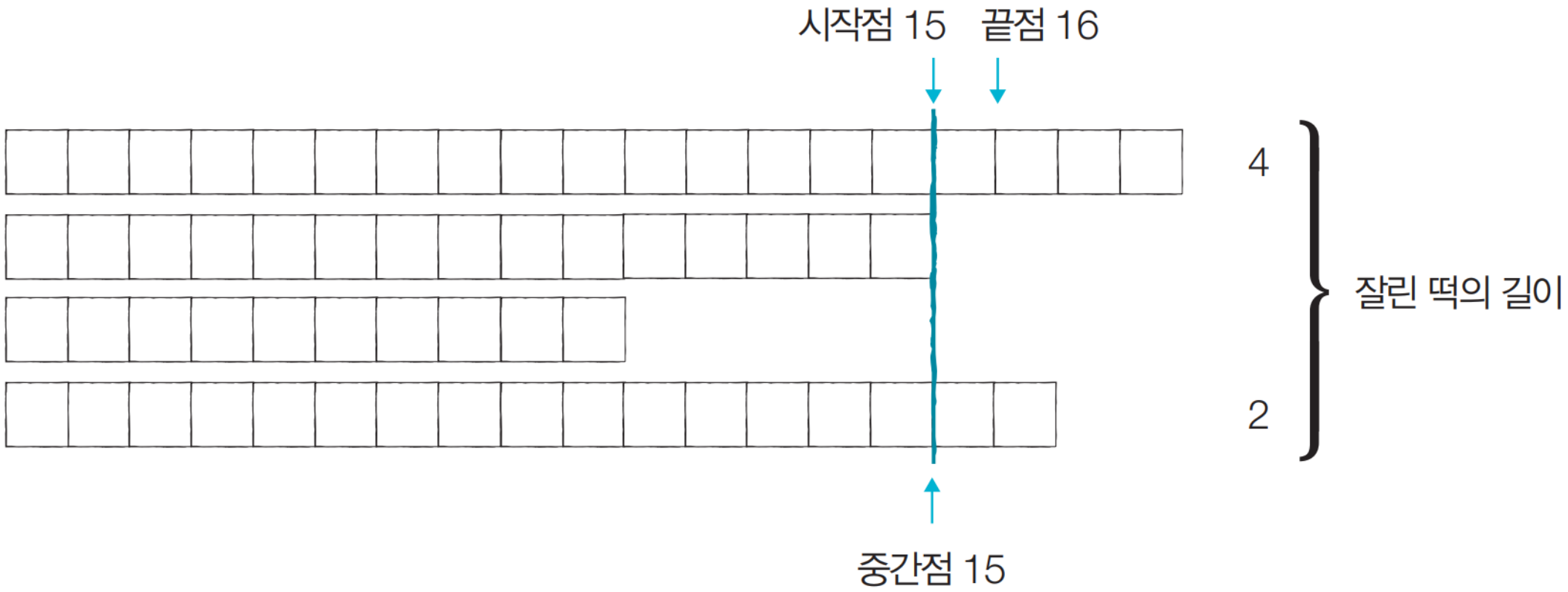
- [Step 2] 시작점: 10, 끝점: 19, 중간점: 14　　이때 필요한 떡의 크기: M = 6이므로, 결과 저장



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



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



떡볶이 떡 만들기: 답안 예시 (Python)

```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> 정렬된 배열에서 특정 수의 개수 구하기: 문제 설명

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

입력 조건

- 첫째 줄에 N과 x가 정수 형태로 공백으로 구분되어 입력됩니다.
  $(1 \le N \le 1{,}000{,}000)$, $(-10^9 \le x \le 10^9)$
- 둘째 줄에 N개의 원소가 정수 형태로 공백으로 구분되어 입력됩니다.
  $(-10^9 \le$ 각 원소의 값 $\le 10^9)$

출력 조건

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

입력 예시 1

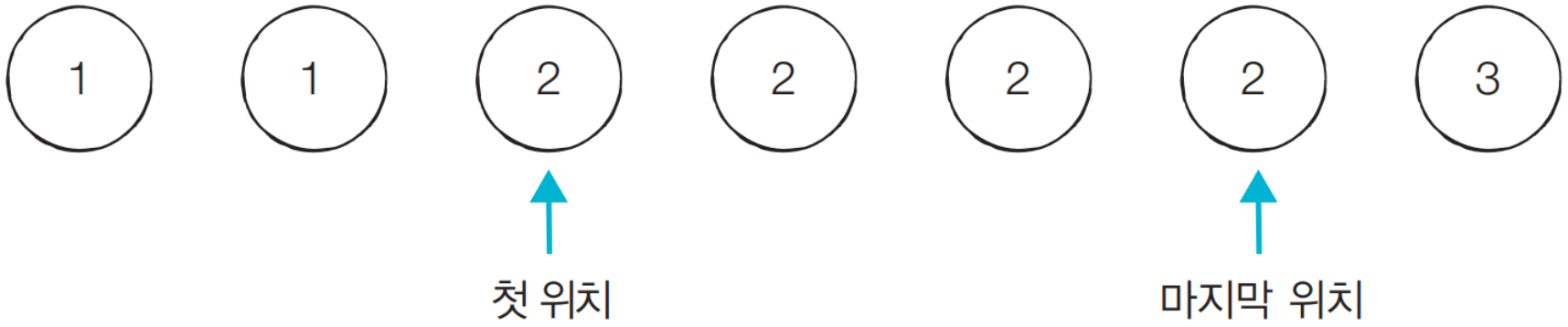
```
7 2
1 1 2 2 2 2 3
```

출력 예시 1

```
4
```

아이디어

- 시간 복잡도 $O(logN)$으로 동작하는 알고리즘을 요구
  - 일반적인 선형 탐색(Linear Search)로는 시간 초과 판정
  - 하지만 데이터가 정렬되어 있기 때문에 이진 탐색을 수행
- 특정 값이 등장하는 첫 번째 위치와 마지막 위치를 찾아 위치 차이를 계산해 문제를 해결



답안 예시 (Python)

```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)
```

실습: 제곱근 구하기

```
def find_sqrt_bin_search(n, error=0.001):
    lower = n < 1 and n or 1
    upper = n < 1 and 1 or n
    mid = lower + (upper - lower) / 2.0
    square = mid * mid
    while abs(square - n) > error:
        if square < n:
            lower = mid
        else:
            upper = mid
        mid = lower + (upper - lower) / 2.0
        square = mid * mid
    return mid
```

```
if __name__ == "__main__":
    a = 2
    b = 9
    import math
    print(math.sqrt(a))
    print(find_sqrt_bin_search(a))
    print(math.sqrt(b))
    print(find_sqrt_bin_search(b))
```