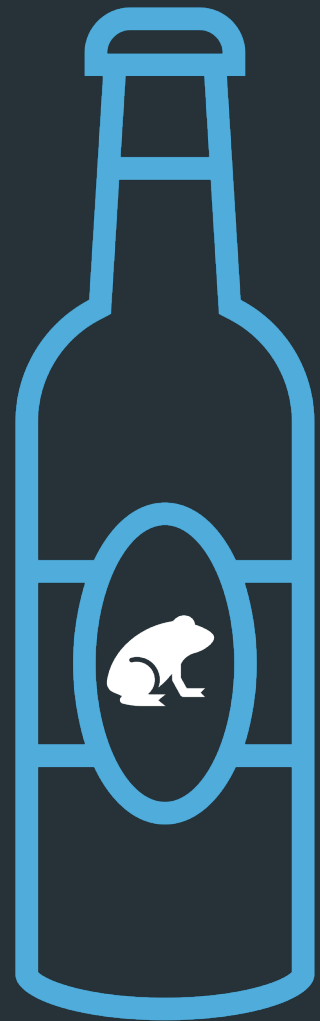


# 알튜비튜

## 이분 탐색

문제의 크기를 절반으로 줄이면서 빠르게 답을 찾는 알고리즘입니다.  
코딩 테스트에서 주로 효율성을 보는 문제에 활용됩니다.



DOWN

40?

5?

UP

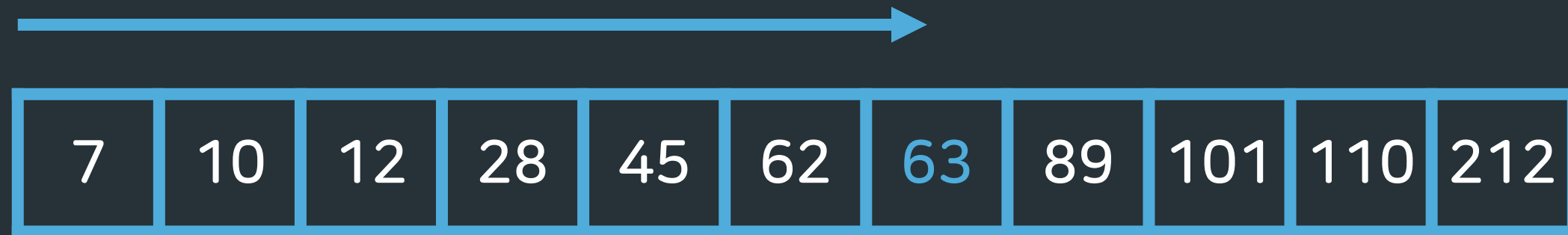
14?

이런 문제가 있다고 해봅시다.

7	10	12	28	45	62	63	89	101	110	212
---	----	----	----	----	----	----	----	-----	-----	-----

“이 안에 63이 있나?”

이런 문제가 있다고 해봅시다.

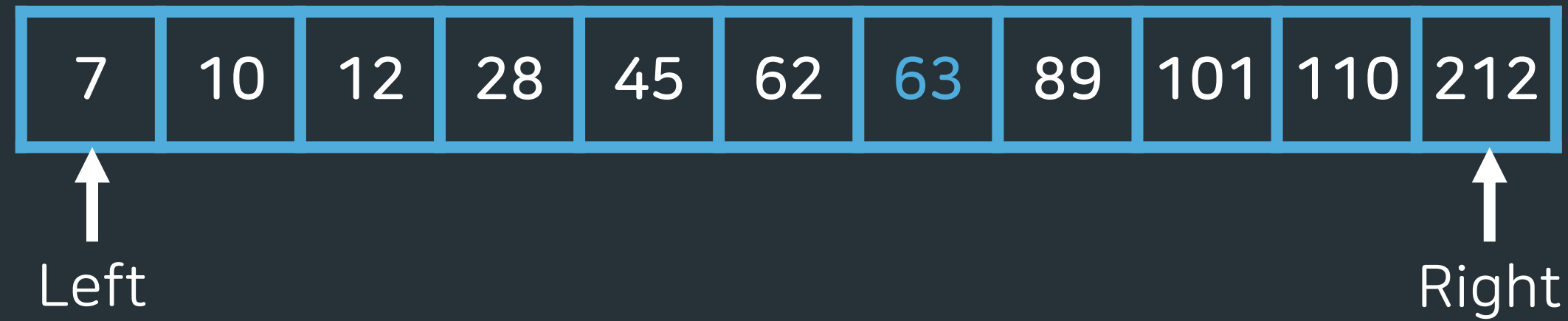


$O(n)$ 으로 찾을 순 있지만...

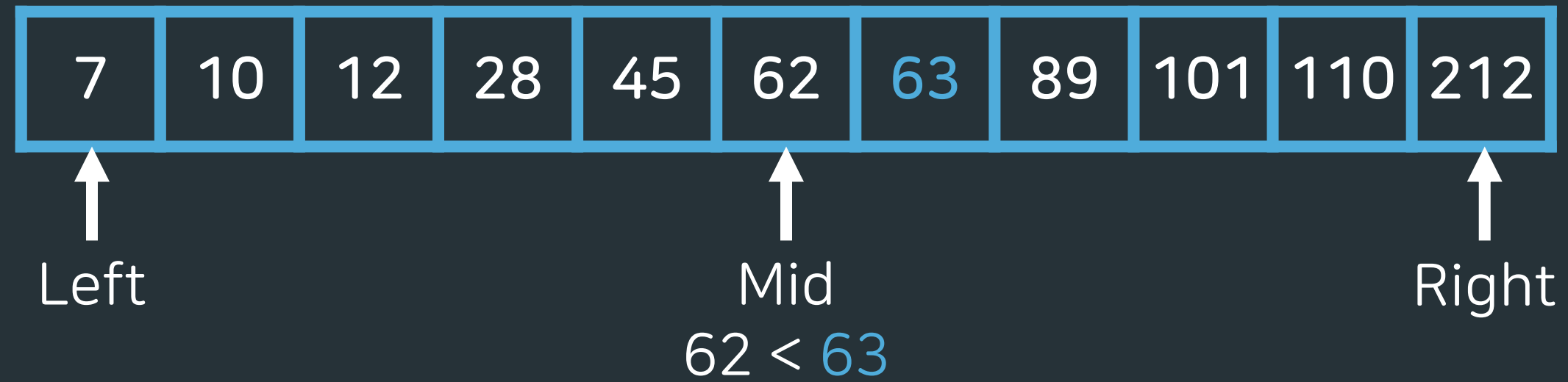
## Binary Search

- 배열의 크기를 절반으로 줄이면서 답을 찾는 알고리즘
- 주로 반복문으로 구현
- 시간 복잡도는  $O(\log n)$
- 입력 범위  $N$ 이 큰 편
- 알고리즘이 성립하기 위해선 배열이 반드시 정렬되어 있어야 함

아까 그 문제를 다시 풀면?

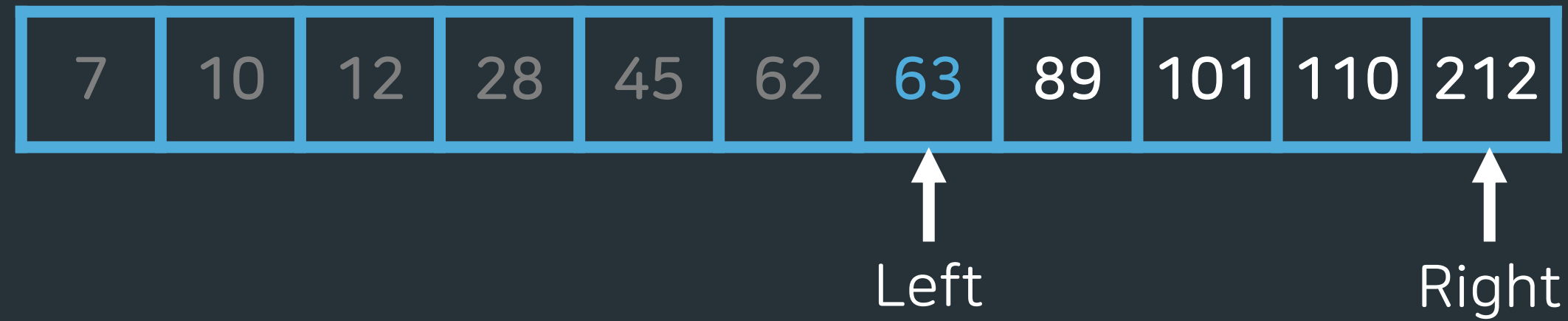


## 아까 그 문제를 다시 풀면?



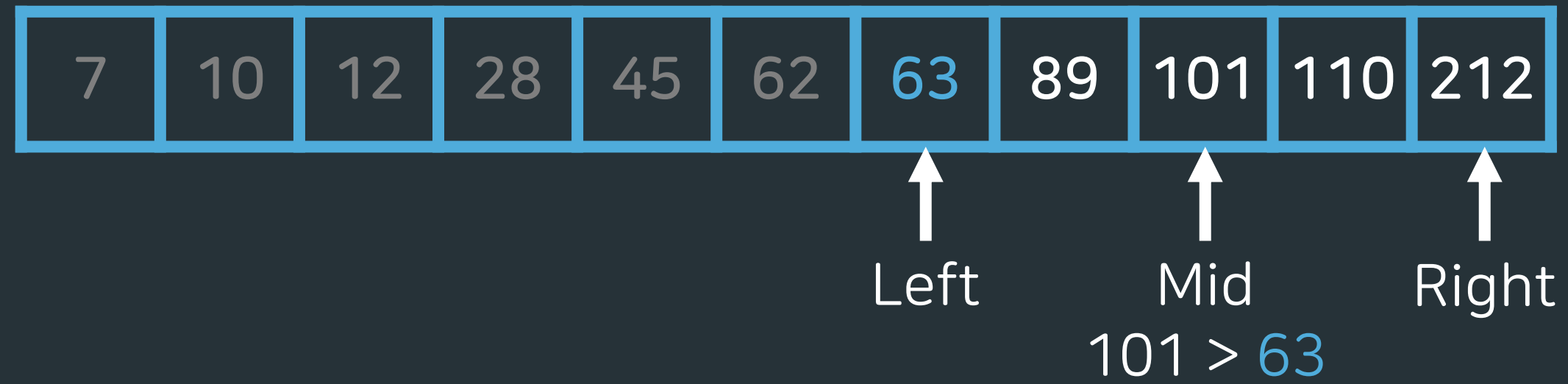
- 연산 횟수: 1

아까 그 문제를 다시 풀면?



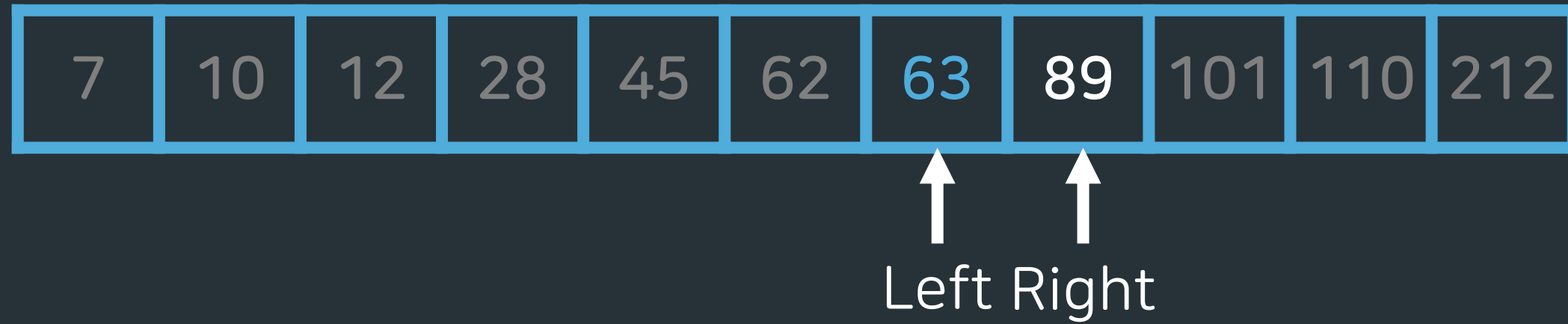


## 아까 그 문제를 다시 풀면?

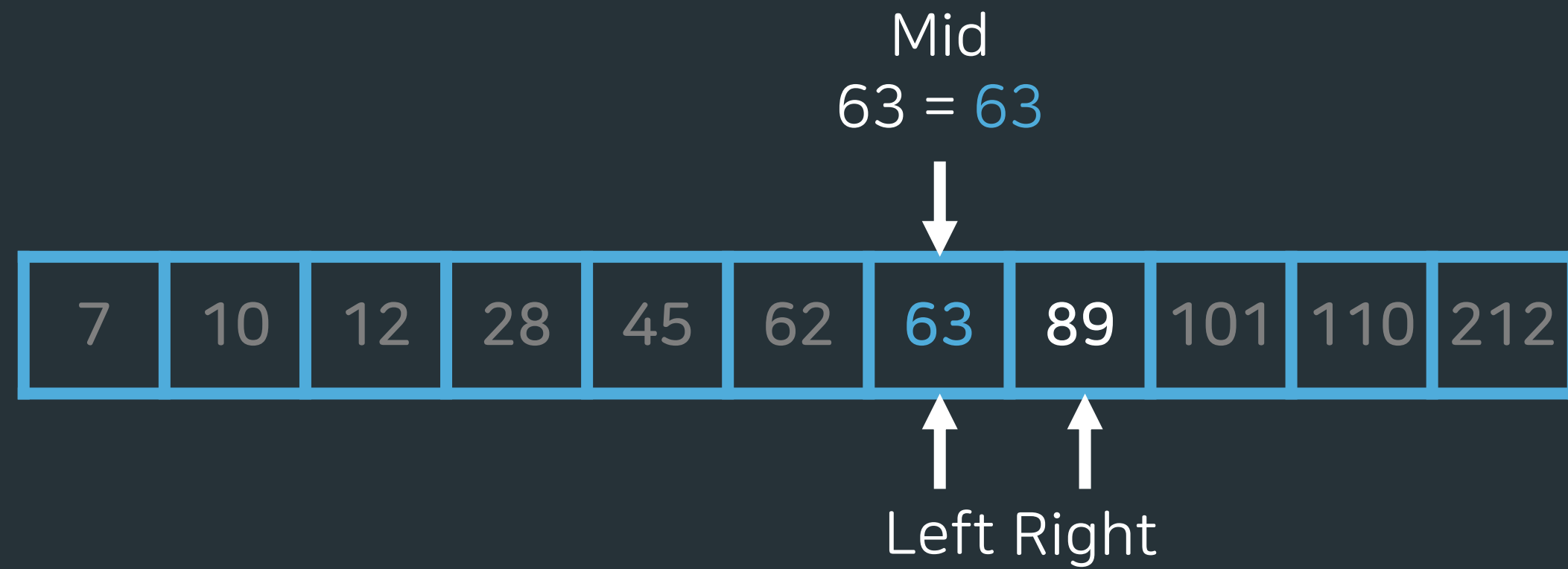


- 연산 횟수: 2

아까 그 문제를 다시 풀면?

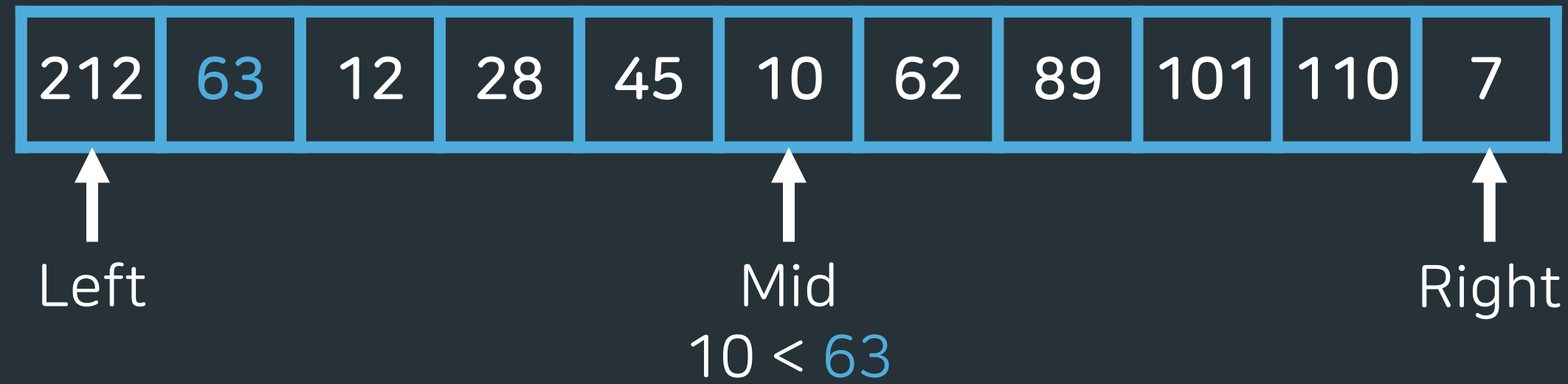


## 아까 그 문제를 다시 풀면?



- 연산 횟수: 3

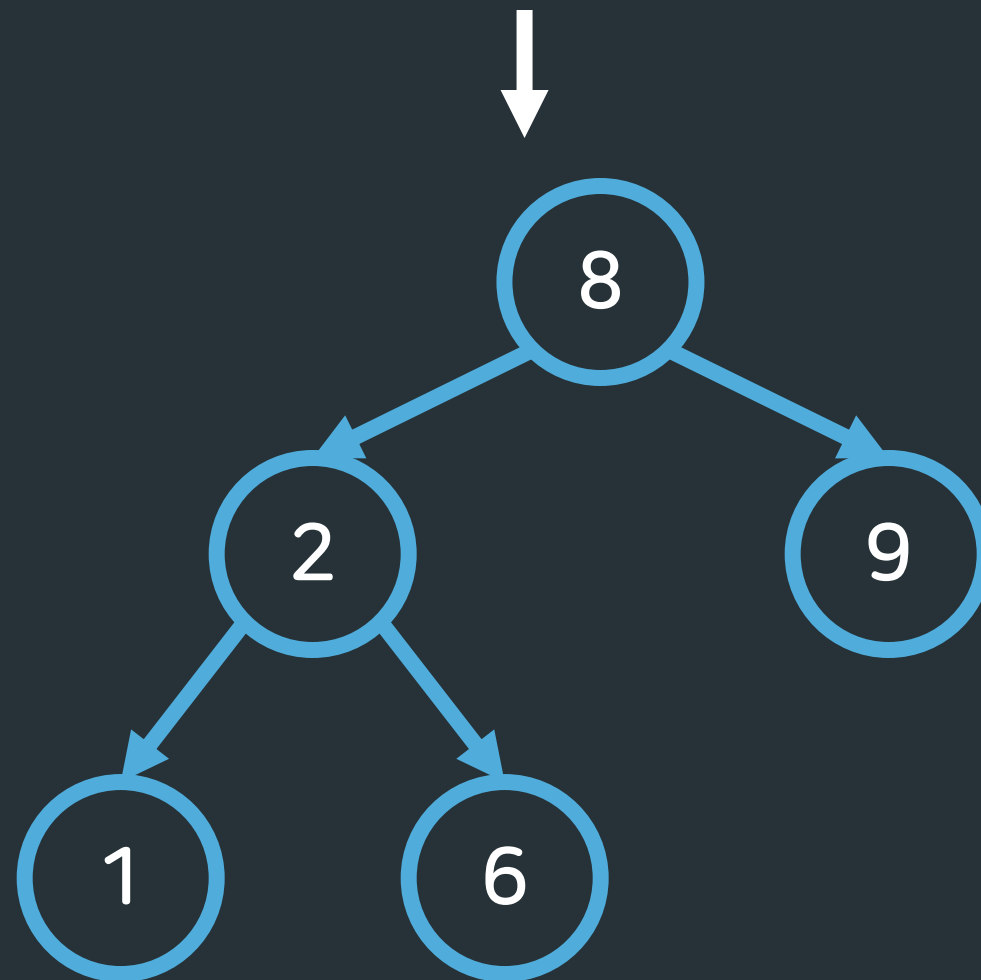
# 만약 정렬이 안 되어 있다면?



# 만약 정렬이 안 되어 있다면?



## ■ 원가 낮설지 않은 이 기분



## ■ 이진 탐색 트리(Binary Search Tree)

- 효율적인 탐색을 위한 이진트리 기반의 자료구조
- $\text{key}(\text{왼쪽서브트리}) < \text{key}(\text{루트노드}) < \text{key}(\text{오른쪽서브트리})$

- 이분 탐색의 대상이 되는 원소들을 **트리**에 넣으면 **BST!**
- BST를 **중위 순회(inorder)**하면 배열이 나옴

## /<> 1920번 : 수 찾기 - Silver 4

### 문제

- N개의 정수가 주어질 때, X라는 정수가 존재하는지 알아내라

### 제한 사항

- N은  $1 \leq N \leq 100,000$
- 알아내야 하는 수 M의 개수는  $1 \leq M \leq 100,000$
- 입력되는 정수 k는 int 범위 내에 있음

### 예제 입력

```
5
4 1 5 2 3
5
1 3 7 9 5
```

### 예제 출력

```
1
1
0
0
1
```

## /<> 1920번 : 수 찾기 - Silver 4

### 문제

- N개의 정수가 주어질 때, X라는 정수가 존재하는지 알아내라

### 제한 사항

- N은  $1 \leq N \leq 100,000$
- 알아내야 하는 수 M의 개수는  $1 \leq M \leq 100,000$
- 입력되는 정수 k는 int 범위 내에 있음

1. 순차 탐색? ->  $100,000 * 100,000 = 1e10 (=100억)$  시간초과
2. 셋? ->  $100,000 * 17 + 100,000 * 17 = 34 * 1e5 (=340만)$  메모리면에서 비효율적이거나 가능  
but 만약 정수의 위치를 요구하면 불가능한 방법

### 예제 입력

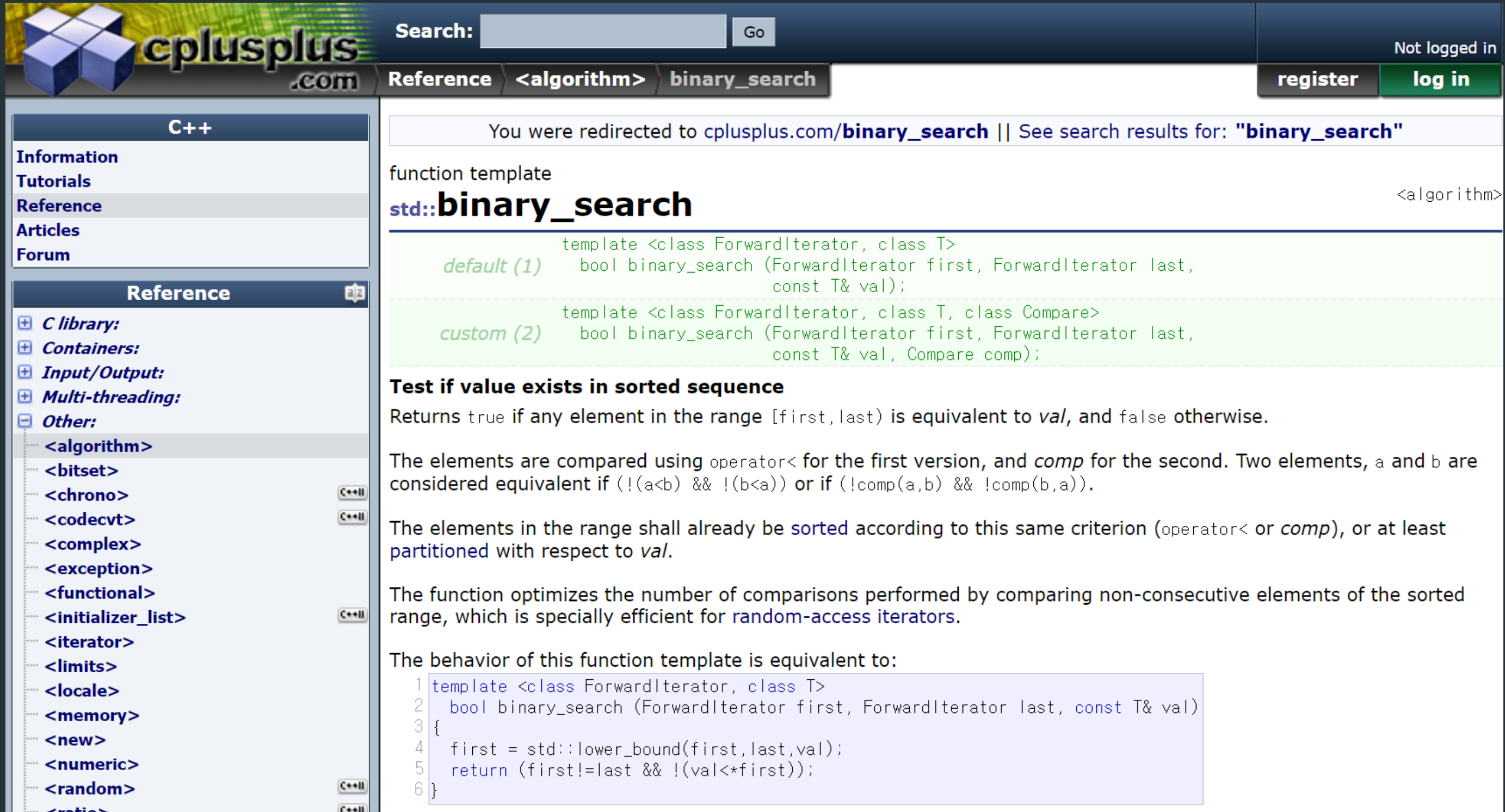
```
5
4 1 5 2 3
5
1 3 7 9 5
```

### 예제

```
1
1
0
0
1
```



# 함수가 있긴 한데요...



The screenshot shows the cplusplus.com website. The top navigation bar includes a search bar, a "Go" button, and links for "Not logged in", "register", and "log in". The main content area displays the "std::binary\_search" function template. It includes two versions: a default version and a custom version with a comparator. Below the code, there is a description of the function's purpose: "Test if value exists in sorted sequence". It explains that the function returns true if any element in the range [first, last) is equivalent to val, and false otherwise. It also notes that the elements are compared using operator< for the first version and comp for the second. The function optimizes the number of comparisons by comparing non-consecutive elements of the sorted range. Finally, it shows the behavior of the function template is equivalent to a specific code snippet.

```
function template
std::binary_search                                     <algorithm>

default (1)      template <class ForwardIterator, class T>
                  bool binary_search (ForwardIterator first, ForwardIterator last,
                                      const T& val);

custom (2)      template <class ForwardIterator, class T, class Compare>
                  bool binary_search (ForwardIterator first, ForwardIterator last,
                                      const T& val, Compare comp);

Test if value exists in sorted sequence
Returns true if any element in the range [first,last) is equivalent to val, and false otherwise.

The elements are compared using operator< for the first version, and comp for the second. Two elements, a and b are
considered equivalent if (!(a<b) && !(b<a)) or if (!comp(a,b) && !comp(b,a)).

The elements in the range shall already be sorted according to this same criterion (operator< or comp), or at least
partitioned with respect to val.

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted
range, which is specially efficient for random-access iterators.

The behavior of this function template is equivalent to:
1 template <class ForwardIterator, class T>
2   bool binary_search (ForwardIterator first, ForwardIterator last, const T& val)
3 {
4   first = std::lower_bound(first, last, val);
5   return (first!=last && !(val<*first));
6 }
```

원소의 존재 여부만 리턴하는 간단한 함수이므로 구현하는 방법을 알아야 함

## /<> 10816번 : 숫자 카드 2 - Silver 4

### 문제

- N개의 카드가 주어진다.
- M개의 정수에 대해 해당 숫자가 적힌 카드가 몇 장인가?

### 제한 사항

- N은  $1 \leq N \leq 500,000$
- 알아내야 하는 수 M의 개수는  $1 \leq M \leq 500,000$
- 입력되는 정수 k는  $-10,000,000 \leq k \leq 10,000,000$

### 예제 입력

```
10
6 3 2 10 10 10 -10 -10 7 3
8
10 9 -5 2 3 4 5 -10
```

### 예제 출력

```
3 0 0 1 2 0 0 2
```

## /<> 10816번 : 숫자 카드 2 - Silver 4

### 문제

- N개의 카드가 주어진다.
- M개의 정수에 대해 해당 숫자가 적힌 카드가 몇 장인가?

### 제한 사항

- N은  $1 \leq N \leq 500,000$
- 알아내야 하는 수 M의 개수는  $1 \leq M \leq 500,000$
- 입력되는 정수 k는  $-10,000,000 \leq k \leq 10,000,000$

1. 순차 탐색? ->  $500,000 * 500,000 = 25 * 10^{10} (=2500억)$  시간초과
2. 셋? ->  $500,000 * 19 + 500,000 * 19 = 19 * 10^6 (=1900만)$  메모리면에서 비효율적이나 가능  
but 만약 배열이 정렬됐을 때, 정수의 범위 인덱스를 요구하면 불가능한 방법

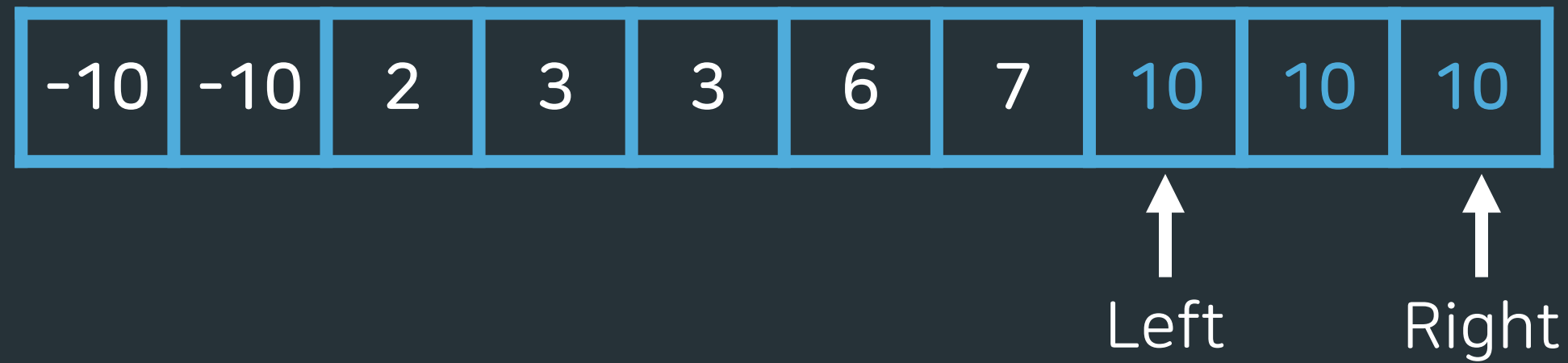
### 예제 입력

```
10
6 3 2 10 10 10 -10 -10 7 3
8
10 9 -5 2 3 4 5 -10
```

### 예제 출력

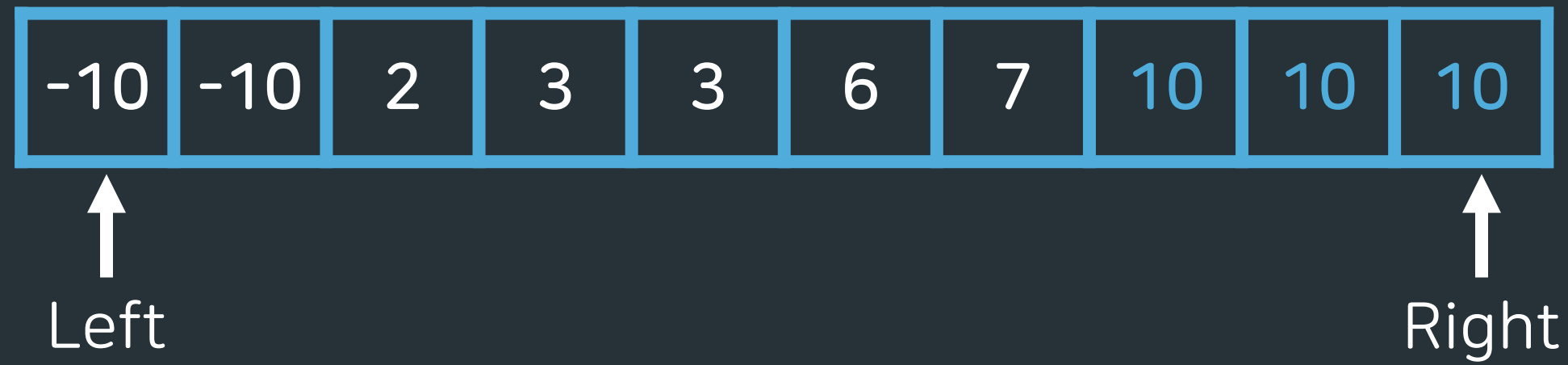
```
3 0 0 1 2 0 0 2
```

## 범위는 어떻게 찾지?

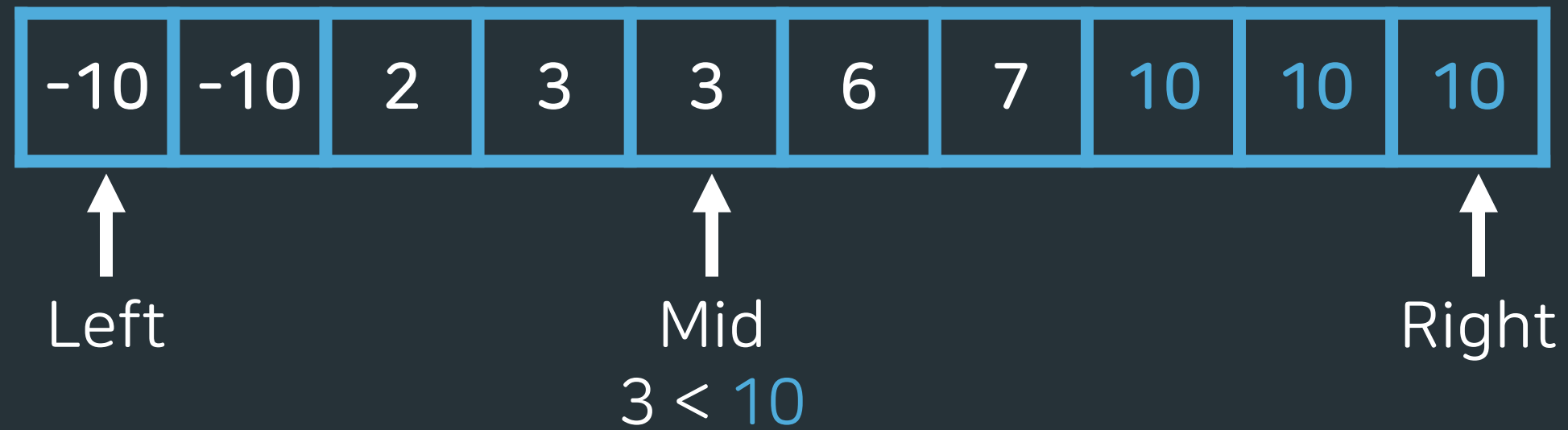


처음으로 10이 나오는 위치와 마지막으로 10이 나오는 위치?

## 처음으로 10이 나오는 위치



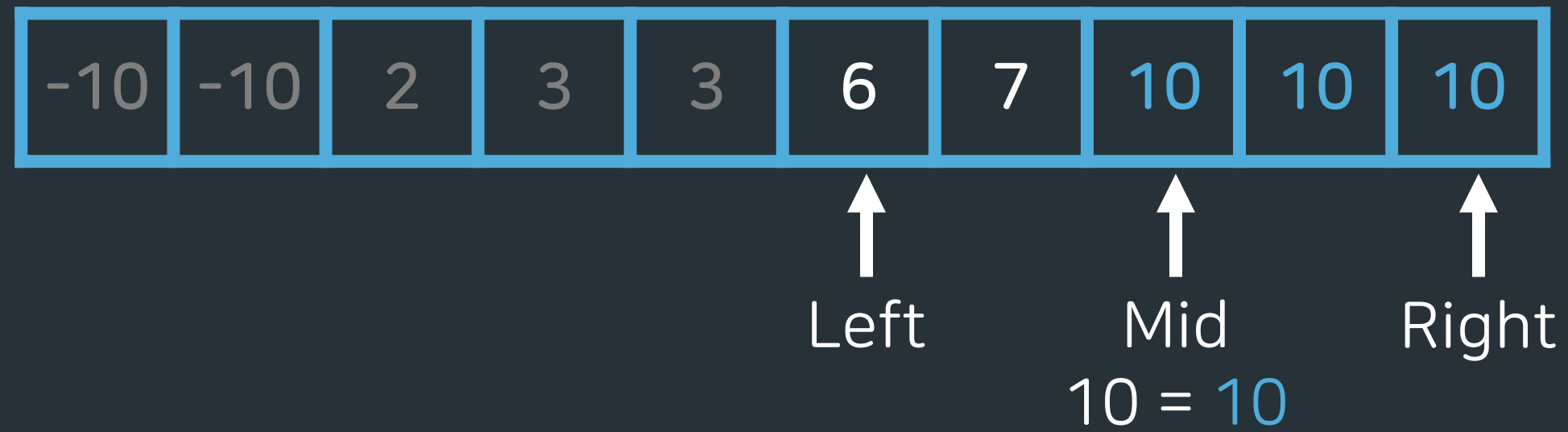
## 처음으로 10이 나오는 위치



## 처음으로 10이 나오는 위치

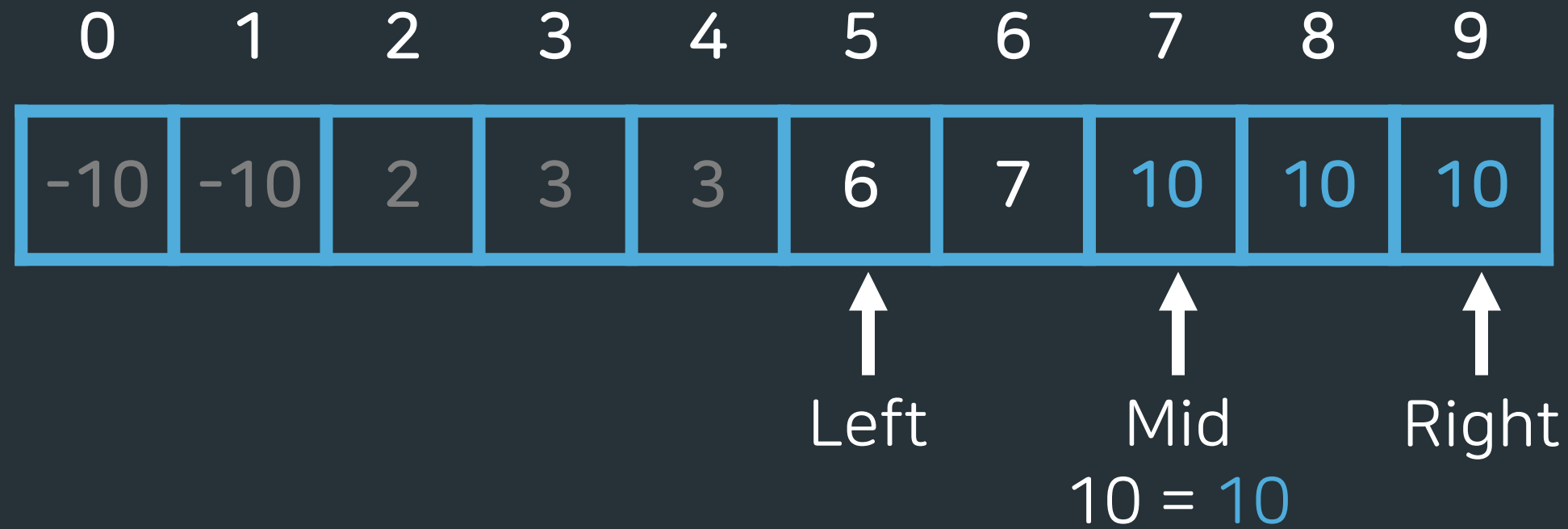


## 처음으로 10이 나오는 위치



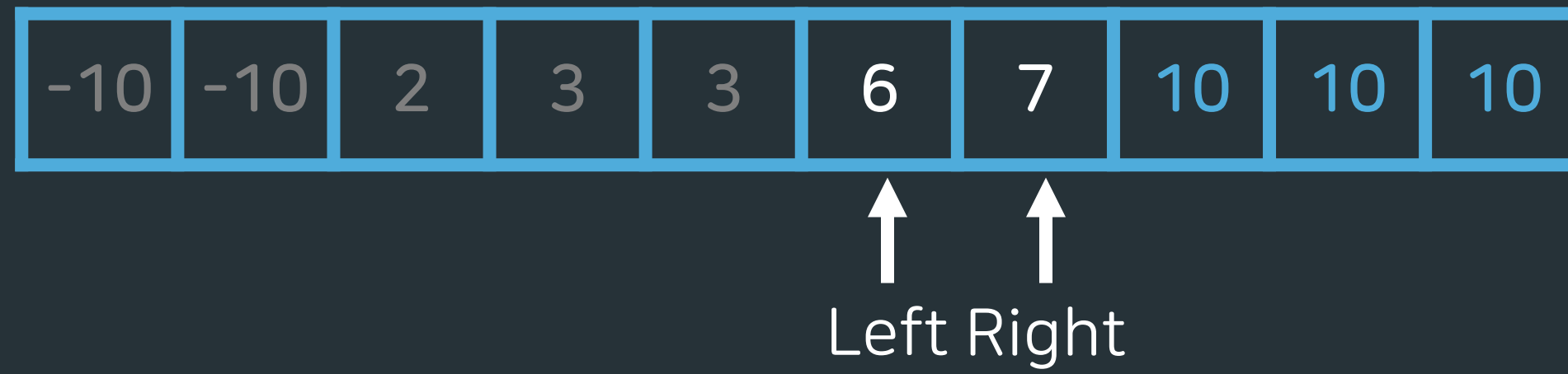


## 처음으로 10이 나오는 위치

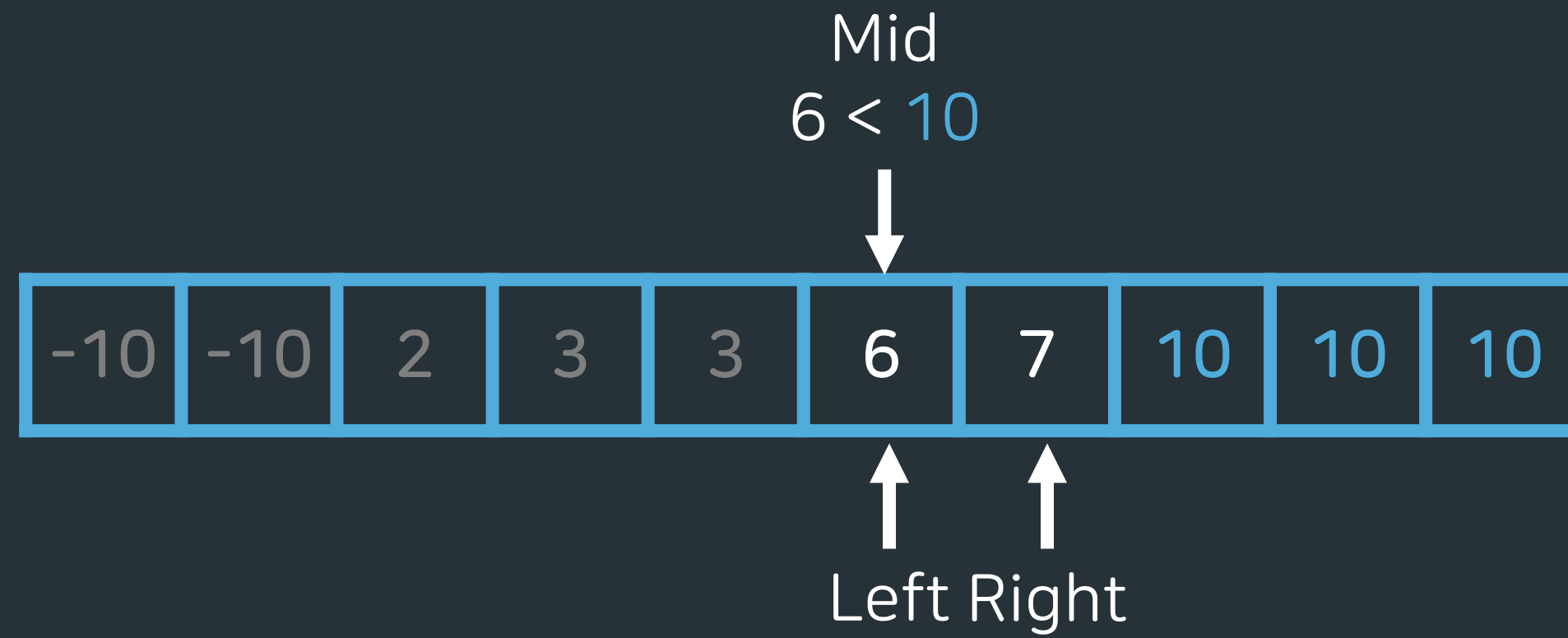


일단 7 위치에 있는데,  
혹시 왼쪽에 10이 더 있지 않을까?

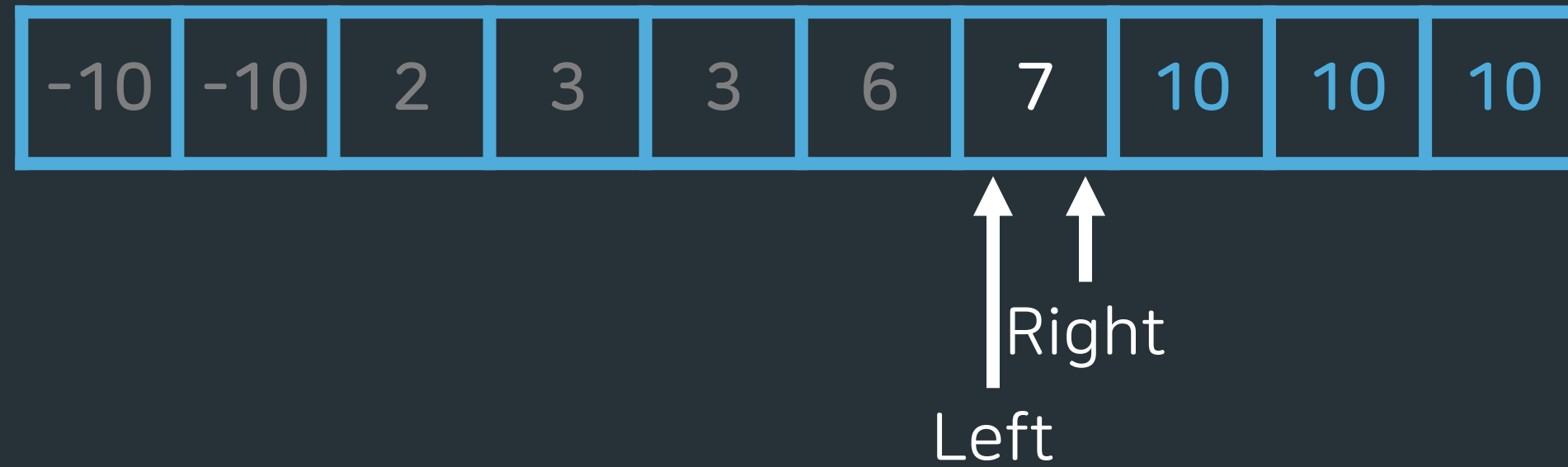
## 처음으로 10이 나오는 위치



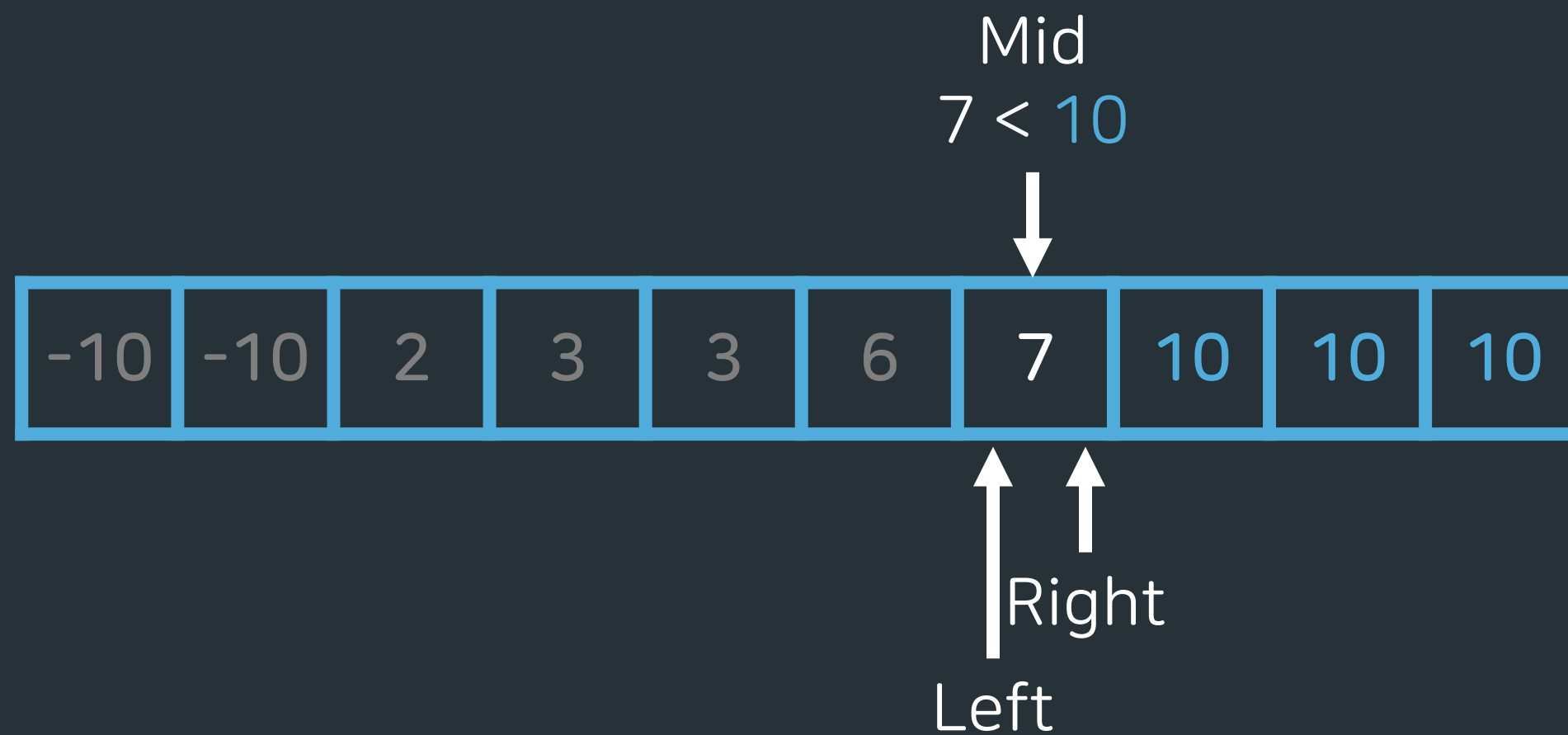
## 처음으로 10이 나오는 위치



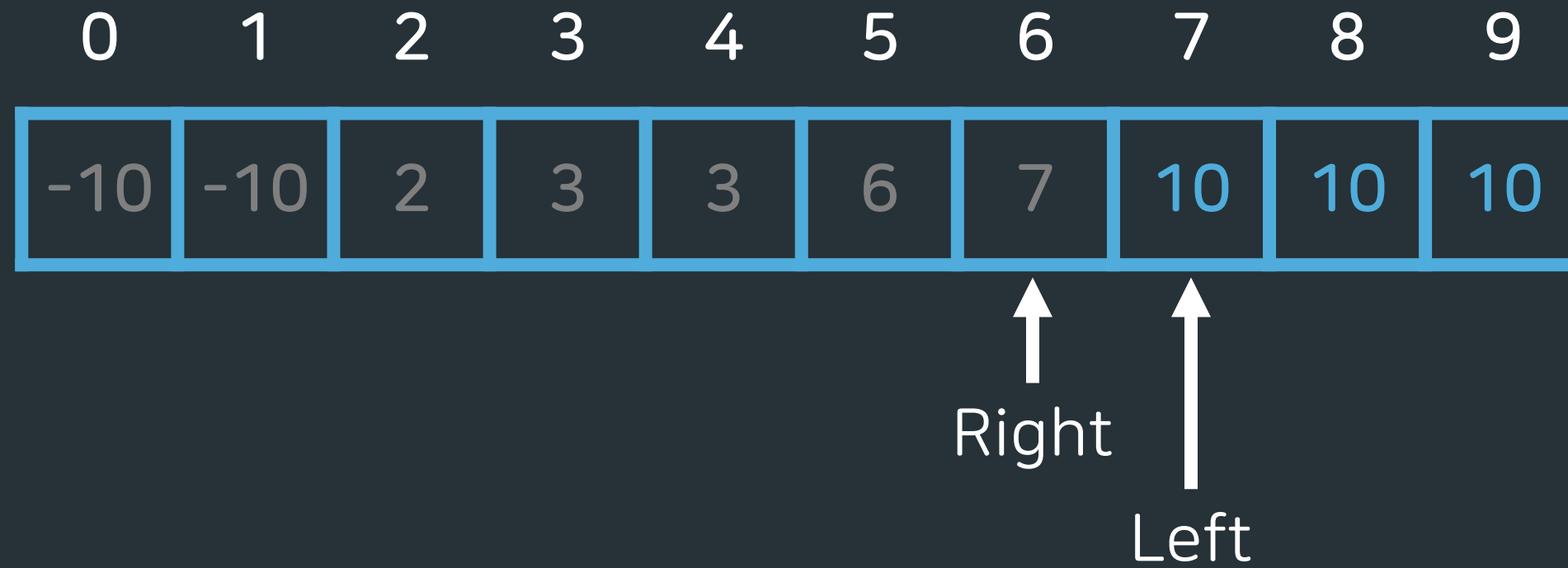
## 처음으로 10이 나오는 위치



# 처음으로 10이 나오는 위치

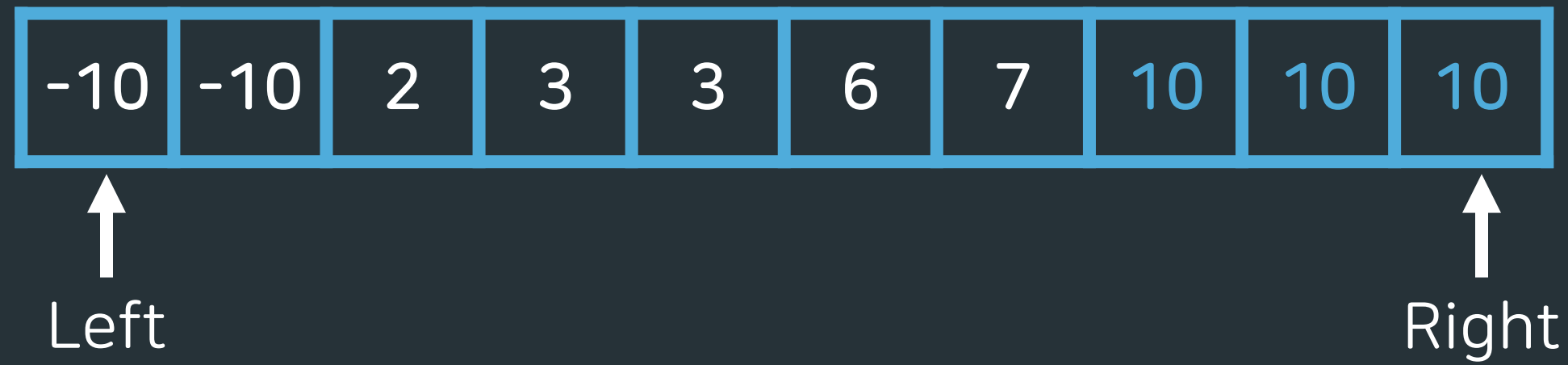


## 처음으로 10이 나오는 위치

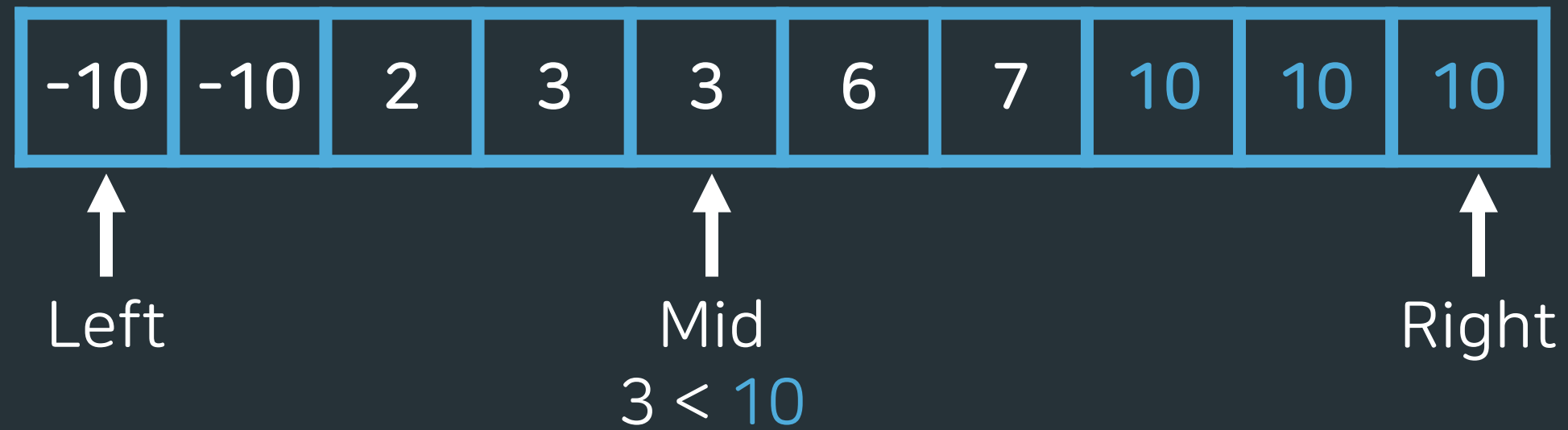


break!  
처음으로 10이 나오는 위치는 7

## 마지막으로 10이 나오는 위치



## 마지막으로 10이 나오는 위치

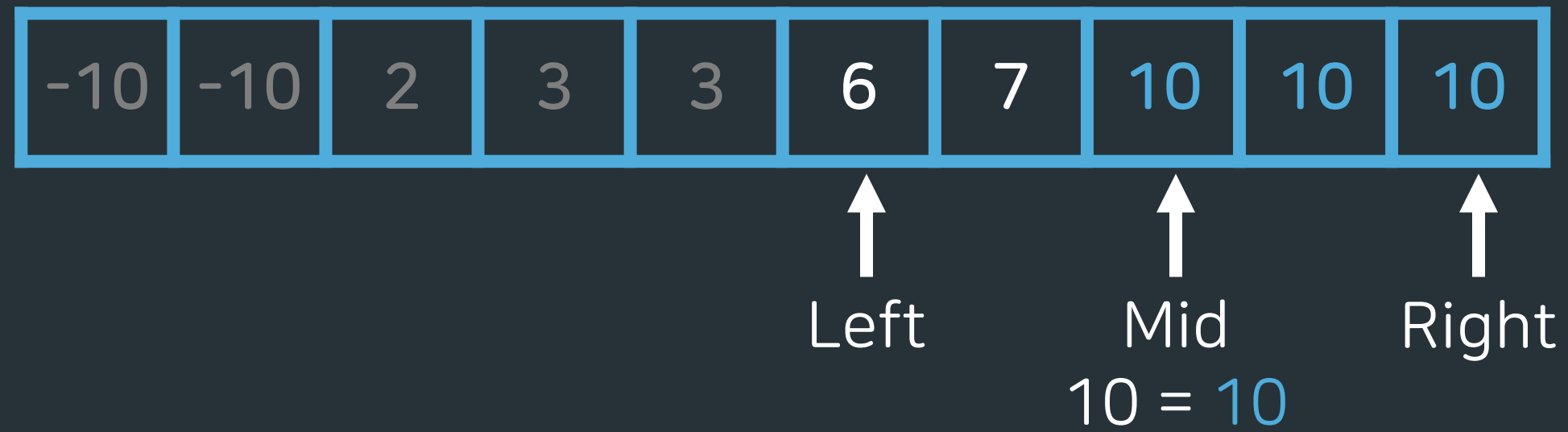




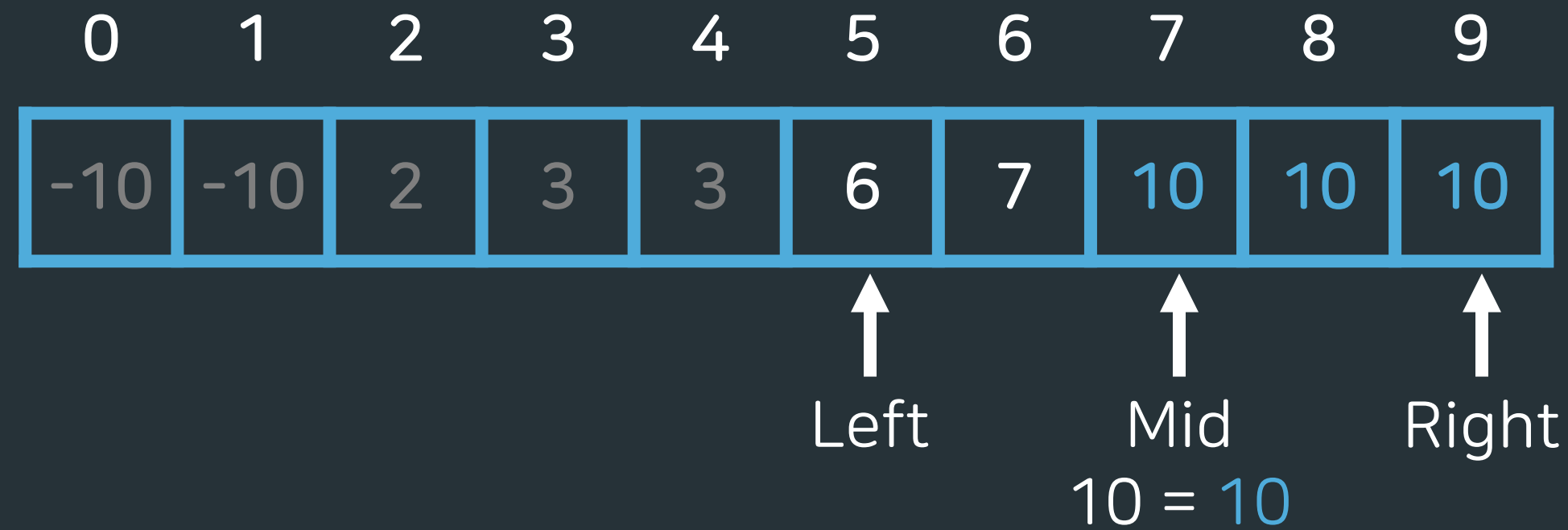
## 마지막으로 10이 나오는 위치



## 마지막으로 10이 나오는 위치

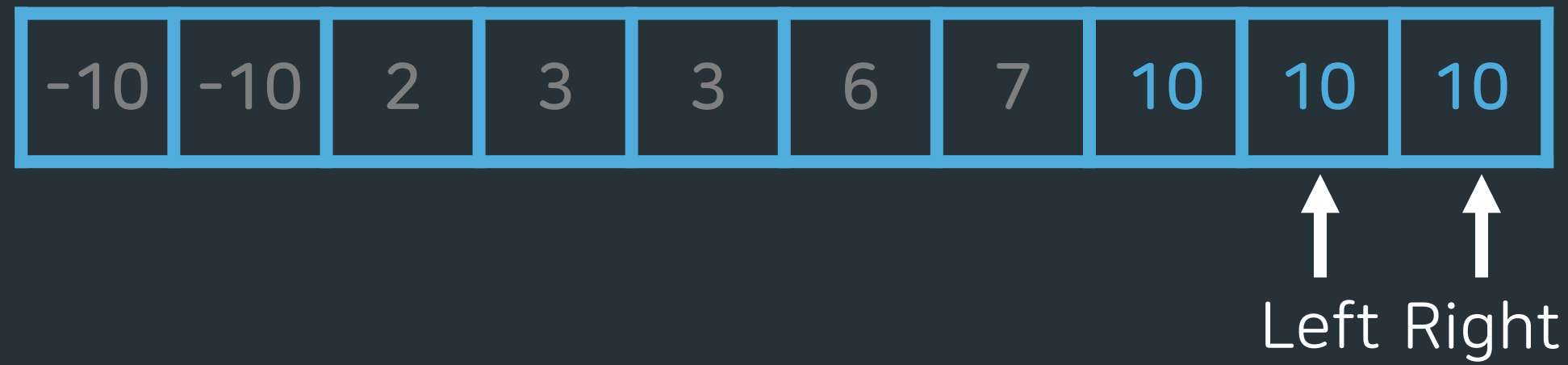


## 마지막으로 10이 나오는 위치

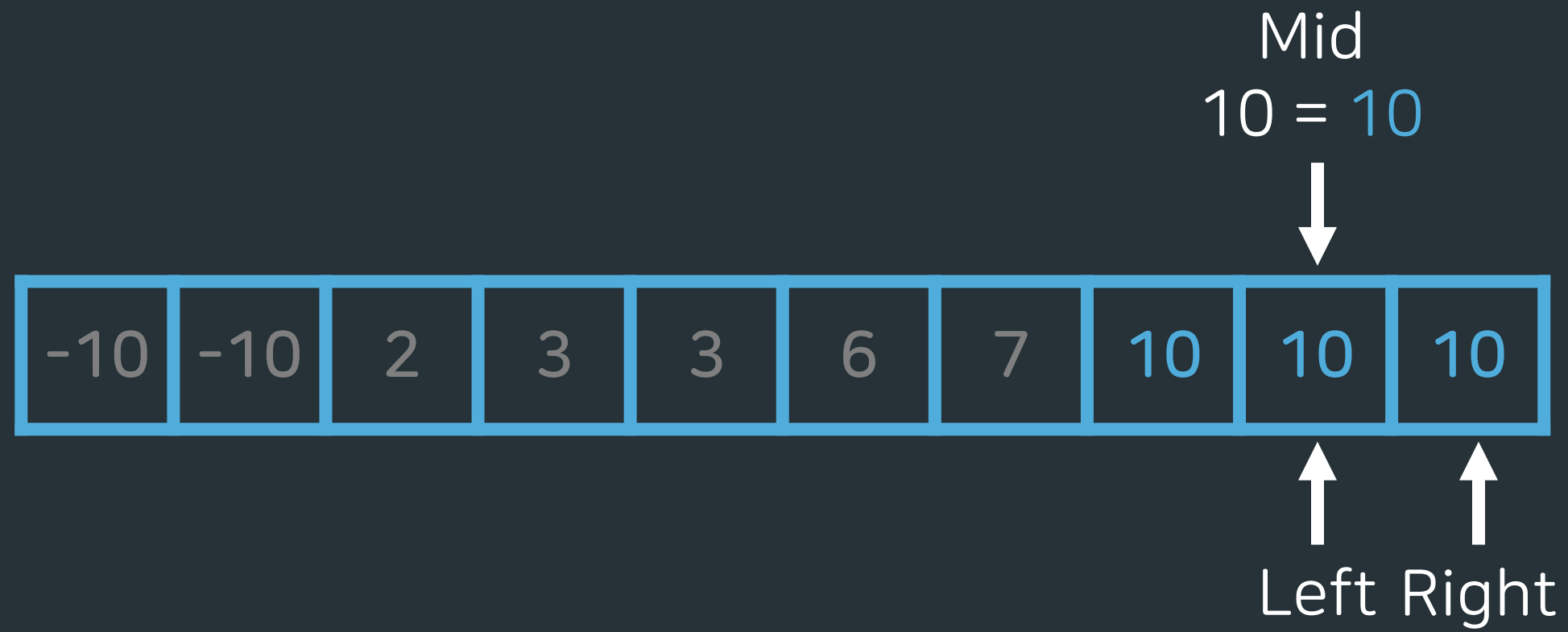


일단 7 위치에 있는데,  
혹시 오른쪽에 10이 더 있지 않을까?

## 마지막으로 10이 나오는 위치



## 마지막으로 10이 나오는 위치



## 마지막으로 10이 나오는 위치

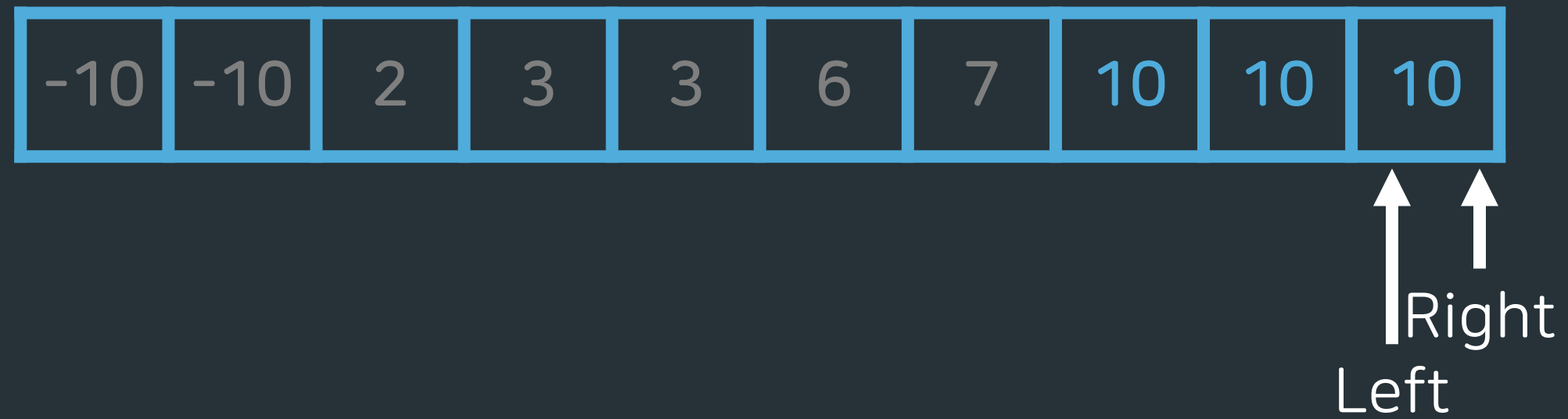
8 위치에 있는데!  
오른쪽으로 더 가볼까?

Mid  
 $10 = 10$

-10	-10	2	3	3	6	7	10	10	10
-----	-----	---	---	---	---	---	----	----	----

↑   ↑  
Left Right

## 마지막으로 10이 나오는 위치

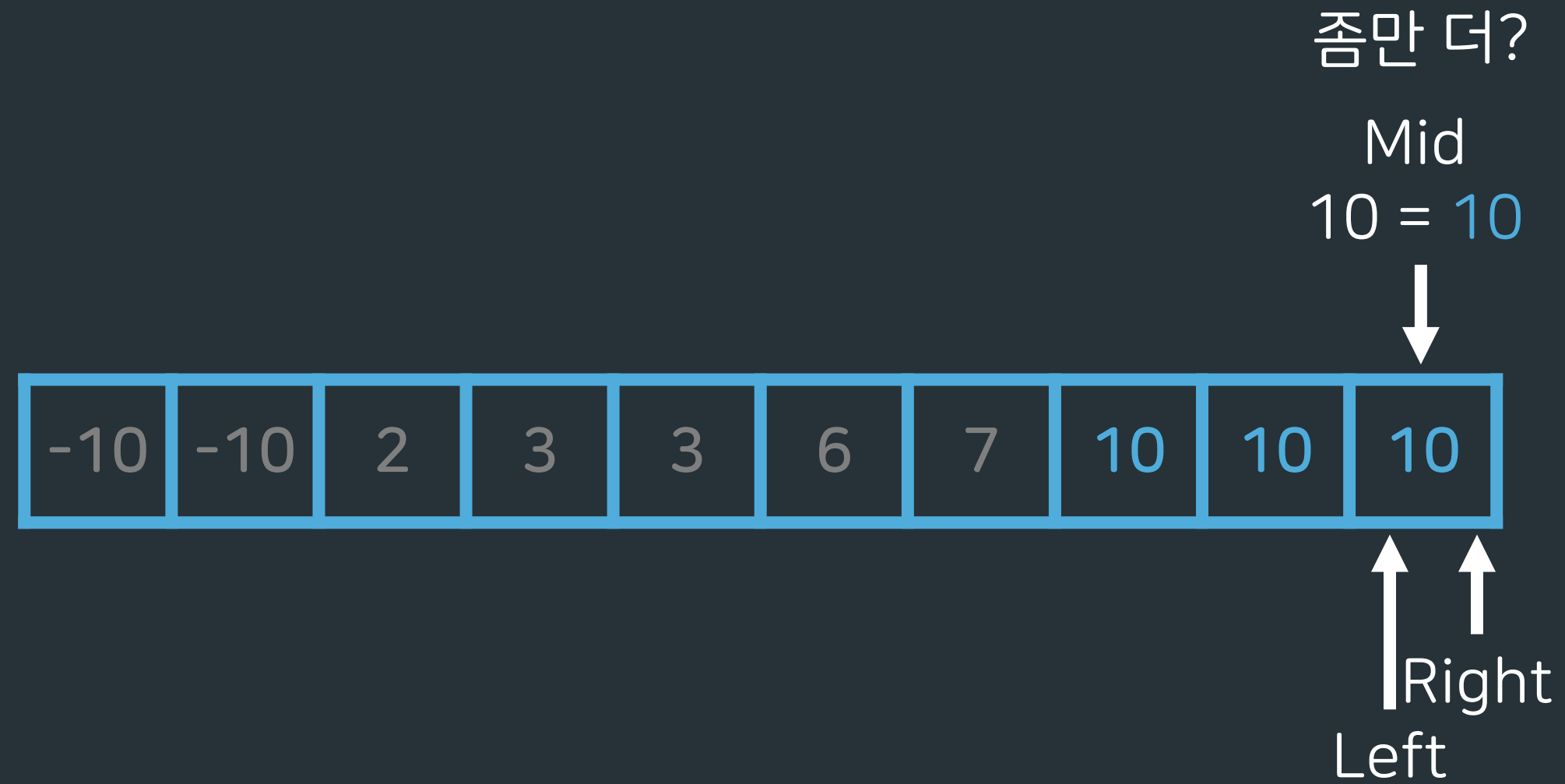


## 마지막으로 10이 나오는 위치

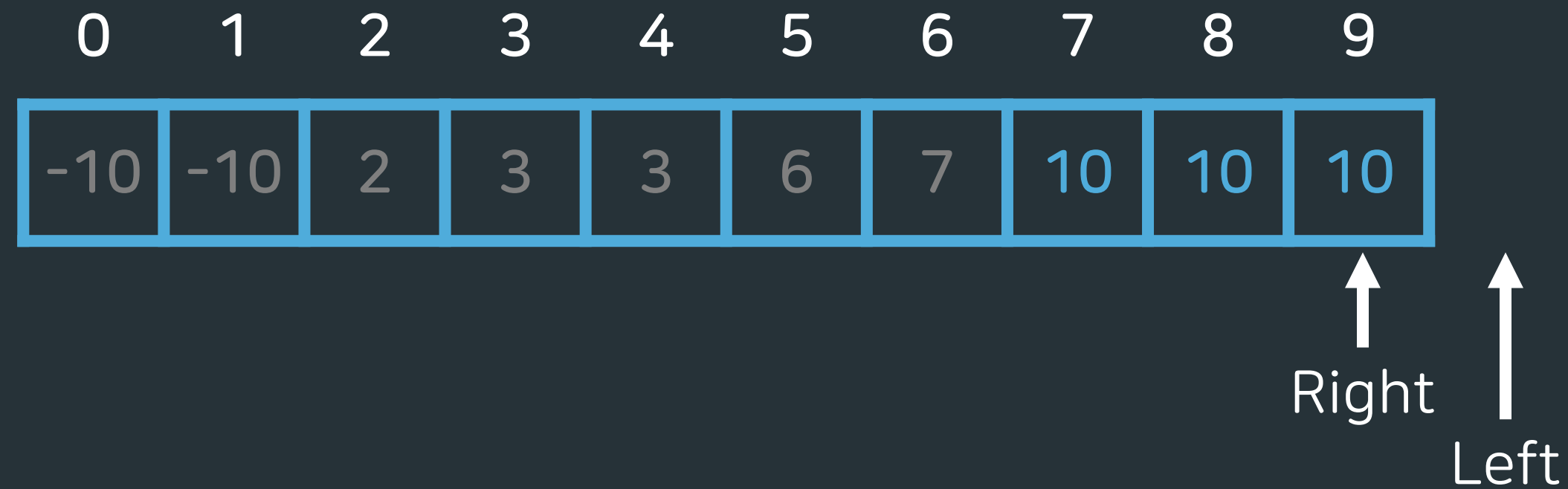




## 마지막으로 10이 나오는 위치



## 마지막으로 10이 나오는 위치



break!  
마지막으로 10이 나오는 위치는 9

0	1	2	3	4	5	6	7	8	9
-10	-10	2	3	3	6	7	10	10	10

처음으로 10이 나오는 위치는 7  
 마지막으로 10이 나오는 위치는 9  
 10의 개수는  $(9-7)+1 = 3$

## Lower Bound

- 찾고자 하는 값인  $X$  이상의 수가 처음으로 나오는 위치

## Upper Bound

- 찾고자 하는 값인  $X$ 를 초과하는 수가 처음으로 나오는 위치

## Lower Bound

- 찾고자 하는 값인  $X$  이상의 수가 처음으로 나오는 위치


## Upper Bound

- 찾고자 하는 값인  $X$ 를 초과하는 수가 처음으로 나오는 위치

\* 정확히 말하면 10의 lower bound는 7이지만, 10의 upper bound는 10

## 함수가 있긴 한데요...!





Search:

Go

Not logged in

register

log in

Reference

<algorithm>

lower\_bound

C++

Information

Tutorials

Reference

Articles

Forum

Reference

C library:

Containers:

Input/Output:

Multi-threading:

Other:

<algorithm>

<bitset>

<chrono>

<codecvt>

<complex>

<exception>

<functional>

<initializer\_list>

<iterator>

<limits>

<locale>

<memory>

<new>

<numeric>

<random>

<ratio>

You were redirected to [cplusplus.com/lower\\_bound](http://cplusplus.com/lower_bound) || See search results for: "lower\_bound"

function template

std::lower\_bound

default (1)

template <class ForwardIterator, class T>  
ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last, const T& val);

custom (2)

template <class ForwardIterator, class T, class Compare>  
ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);

Return iterator to lower bound

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than *val*.

The elements are compared using `operator<` for the first version, and `comp` for the second. The elements in the range shall already be [sorted](#) according to this same criterion (`operator<` or `comp`), or at least [partitioned](#) with respect to *val*.

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for [random-access](#) iterators.

Unlike [upper\\_bound](#), the value pointed by the iterator returned by this function may also be equivalent to *val*, and not only greater.

The behavior of this function template is equivalent to:

```

1 template <class ForwardIterator, class T>
2 ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& val)
3 {
4     ForwardIterator it;
5     iterator_traits<ForwardIterator>::difference_type count, step;
6     count = distance(first,last);
7     while (count>0)

```

C++

Information

Tutorials

Reference

Articles

Forum

Reference

C library:

Containers:

Input/Output:

Multi-threading:

Other:

<algorithm>

<bitset>

<chrono>

<codecvt>

<complex>

<exception>

<functional>

<initializer\_list>

<iterator>

<limits>

<locale>

<memory>

<new>

<numeric>

<random>

<ratio>

You were redirected to [cplusplus.com/upper\\_bound](http://cplusplus.com/upper_bound) || See search results for: "upper\_bound"

function template

std::upper\_bound

default (1)

template <class ForwardIterator, class T>  
ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last, const T& val);

custom (2)

template <class ForwardIterator, class T, class Compare>  
ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);

Return iterator to upper bound

Returns an iterator pointing to the first element in the range [first,last) which compares greater than *val*.

The elements are compared using `operator<` for the first version, and `comp` for the second. The elements in the range shall already be [sorted](#) according to this same criterion (`operator<` or `comp`), or at least [partitioned](#) with respect to *val*.

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for [random-access](#) iterators.

Unlike [lower\\_bound](#), the value pointed by the iterator returned by this function cannot be equivalent to *val*, only greater.

The behavior of this function template is equivalent to:

```

1 template <class ForwardIterator, class T>
2 ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last, const T& val)
3 {
4     ForwardIterator it;
5     iterator_traits<ForwardIterator>::difference_type count, step;
6     count = std::distance(first,last);
7     while (count>0)
8     {

```

이분 탐색 문제의 특징 때문에 구현하는 방법을 알아야 함

## /<> 2110번 : 공유기 설치 - Gold 4

### 문제

- N개의 집이 수직선상에 놓여있다.
- 공유기는 C개가 있고, 한 집에 최대 하나만 설치할 수 있다.
- 공유기 C개를 설치할 때, 가장 인접한 두 공유기 사이의 최대 거리는?

### 제한 사항

- N의 범위는  $2 \leq N \leq 200,000$
- C의 범위는  $2 \leq C \leq N$
- 집의 좌표 x의 범위는  $0 \leq x \leq 1,000,000,000$

### 예제 입력

```
5 3
1
2
8
4
9
```

### 예제 출력

```
3
```

## Parametric Search

- 최적화 문제를 결정 문제로 바꾸는 알고리즘
- 가능한 답의 나열이 비내림차순 또는 비오름차순이어야 함
- 최댓값, 최솟값을 요구하는 문제라면 Parametric Search가 아닌지 의심!
- 탐색을 시작할 `left, right`를 잘 정해야 함



공유기 C개를 설치할 때, 가장 인접한 두 공유기 사이의 최대 거리는?

공유기 C개를 설치할 때, 가장 인접한 두 공유기 사이의 최대 거리는?  
-> 가장 인접한 두 공유기 사이의 거리가  $K$ 일 때, 공유기 C개를 설치할 수 있는가?

질문에 대한 답이 하나로 나오는 결정 문제로 바꾸기  
가능한 모든  $K$  중 최댓값이 두 공유기 사이의 최대거리

	13	14	15	16	17	18	19	20	21	22
arr	2	2	3	3	3	3	3	4	4	4

$arr[i] = k$  (=i원을 동전으로 나타낼 때, 필요한 5원짜리의 개수는 k)

5원짜리 동전 3개와 5원 이하의 동전들로 나타낼 수 있는 **최댓값**은?

\*단, 남은 금액을 5로 나눈 값이 1이상이라면 반드시 5원을 사용해야 한다.

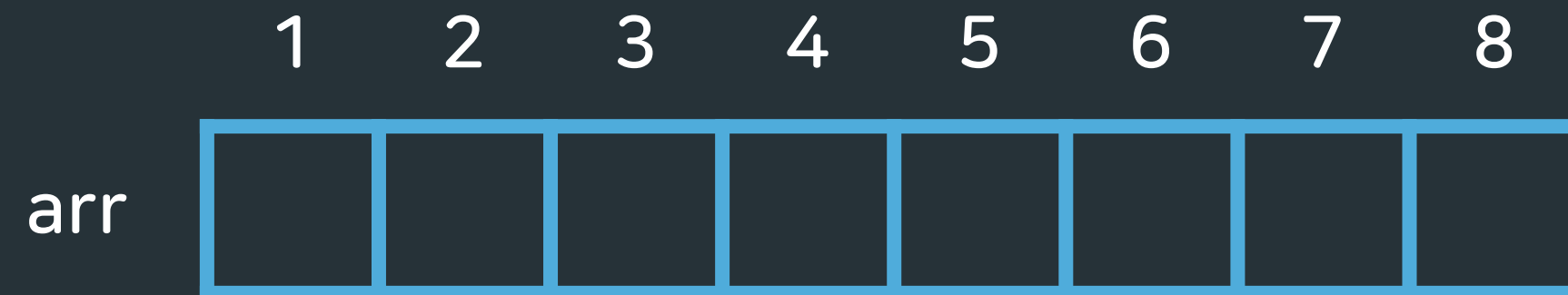
	13	14	15	16	17	18	19	20	21	22
arr	5	6	3	4	5	6	7	4	5	6

$arr[i] = k$  (=i원에 쓰이는 동전은 총 k개)

5원짜리 동전과 1원짜리 동전 총 4개를 사용하여 나타낼 수 있는 **최댓값**은?

\*단, 남은 금액을 5로 나눈 값이 1이상이라면 반드시 5원을 사용해야 한다.

# 우리가 풀게 될 문제는?



$arr[i] = k$  (=가장 인접한 공유기 사이의 거리가  $i$  이상일 때 설치되는 최대 공유기의 개수)

# 우리가 풀게 될 문제는?



	1	2	3	4	5	6	7	8
arr	5							

$arr[i] = k$  (=가장 인접한 공유기 사이의 거리가  $i$ 이상일 때 설치되는 최대 공유기의 개수)

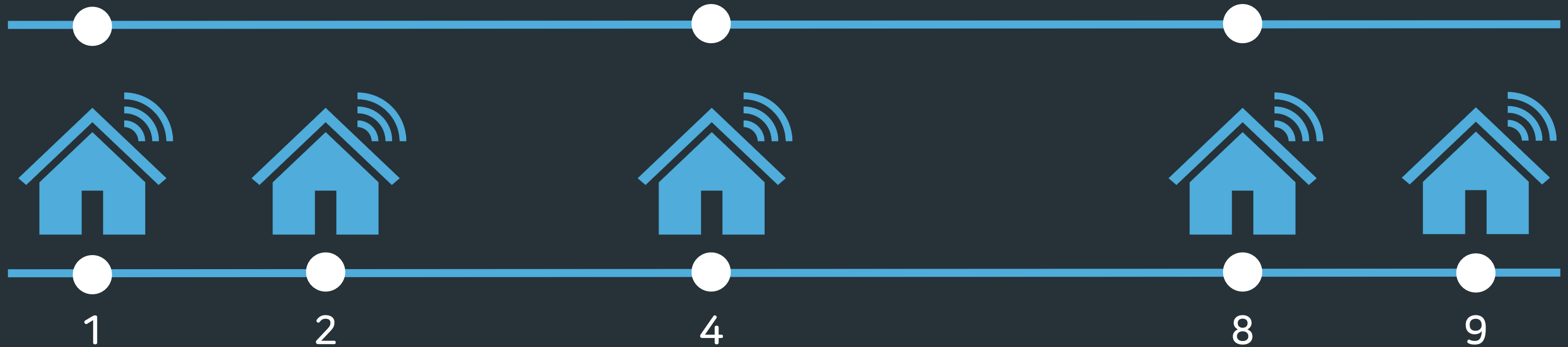
# 우리가 풀게 될 문제는?



	1	2	3	4	5	6	7	8
arr	5	3						

$arr[i] = k$  (=가장 인접한 공유기 사이의 거리가  $i$  이상일 때 설치되는 최대 공유기의 개수)

# 우리가 풀게 될 문제는?



	1	2	3	4	5	6	7	8
arr	5	3	3					

$arr[i] = k$  (=가장 인접한 공유기 사이의 거리가  $i$  이상일 때 설치되는 최대 공유기의 개수)



# 우리가 풀게 될 문제는?



	1	2	3	4	5	6	7	8
arr	5	3	3	2				

$arr[i] = k$  (=가장 인접한 공유기 사이의 거리가  $i$  이상일 때 설치되는 최대 공유기의 개수)

# 우리가 풀게 될 문제는?



	1	2	3	4	5	6	7	8
arr	5	3	3	2	2	2	2	

$arr[i] = k$  (=가장 인접한 공유기 사이의 거리가  $i$ 이상일 때 설치되는 최대 공유기의 개수)

# 우리가 풀게 될 문제는?



	1	2	3	4	5	6	7	8
arr	5	3	3	2	2	2	2	2

$arr[i] = k$  (=가장 인접한 공유기 사이의 거리가  $i$  이상일 때 설치되는 최대 공유기의 개수)

## /<> 2110번 : 공유기 설치 - Gold 4

### 문제

- N개의 집이 수직선상에 놓여있다.
- 공유기는 C개가 있고, 한 집에 최대 하나만 설치할 수 있다.
- 공유기 C개를 설치할 때, 가장 인접한 두 공유기 사이의 최대 거리는?

### 제한 사항

- N의 범위는  $2 \leq N \leq 200,000$
- C의 범위는  $2 \leq C \leq N$
- 집의 좌표 x의 범위는  $0 \leq x \leq 1,000,000,000$

### 예제 입력

```
5 3
1
2
8
4
9
```

### 예제 출력

```
3
```

## 정리

- 배열의 원소가 정렬된 상태라면 이분 탐색보다 더 빨리 특정 원소를 찾을 수는 없음!  $O(\log n)$
- 분할 정복과 달리 문제를 나눈 뒤, 답이 없을 문제의 절반을 버림
- 코딩 테스트에는 주로 Parametric Search 문제로 등장
- 최댓값, 최솟값이라는 키워드가 보인다면 일단 이분 탐색을 의심해보자
- 효율성 테스트 문제로 아주 많이 출제됨

## 이것도 알아보세요

- (C++)원소를 정렬된 상태로 저장하는 맵과 셋에도 lower, upper bound가 있습니다. 한 번 찾아보세요

## 필수

- /<> 17266번: 어두운 굴다리 - Silver 4
- /<> 10815번 : 숫자 카드 - Silver 5
- /<> 16401번 : 과자 나눠주기 - Silver 2

## 도전

- /<> 2343번 : 기타 레슨 - Silver 1
- /<> 3079: 입국심사 - Gold 5

과제제출 마감

~ 5월 9일 금요일 18:59

추가제출 마감

~ 5월 11일 일요일 23:59