# 21차시

# 데스스타 BOJ 11811

- 원래의 수열을 구하자
- 수열의 각 원소를 $a_1, a_2, \ldots, a_N$이라고 하자
- 행렬이 주어지며 $i \neq j, 1 \leq i, j \leq N$인 $i, j$에 대하여 $a_i$와 $a_j$를 AND 비트연산 한 결과가 주어진다
- $1 \leq N \leq 1\,000$

# 데스스타 BOJ 11811

- 가능한 경우만 입력으로 주어진다
- 따라서 입력을 그대로 사용하면 된다

- 입력에 대한 답이 존재한지 또는 불가능한지 확인하려면 2-SAT 문제로 변질된다

# 데스스타 BOJ 11811

- 만약 $a_i$와 $a_i$를 AND 한 결과가 주어지는 경우 $a_i$를 바로 알 수 있다

- 같은 수를 AND하므로 0과 0을 AND하거나 1과 1을 AND하게 되는데, 0과 0을 AND 하는 경우 0, 1과 1을 AND 하는 경우 1이 나온다.
- A & A = A이므로 바로 값이 나오게 된다

# 데스스타 BOJ 11811

- AND의 연산 결과는 두 숫자 모두 해당 자리에 1을 가지고 있어야 1이 나오게 된다
- $a_i$와 $a_j$를 AND 비트연산한 결과에서 어떠한 자리에 1이 있다면 $a_i$와 $a_j$는 반드시 그 자리에 1을 둘 다 갖고 있다

# 데스스타 BOJ 11811

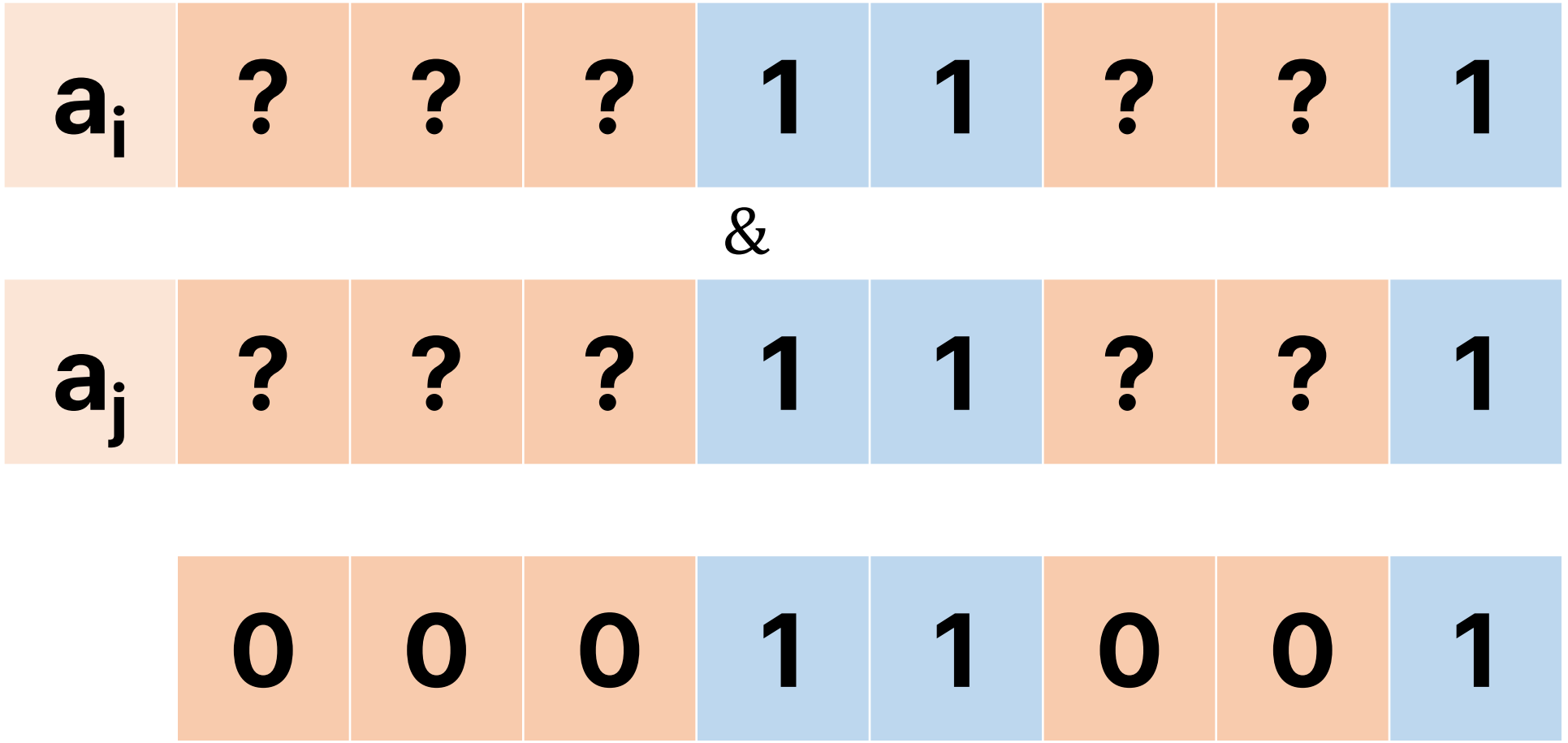
| $a_i$ | ? | ? | ? | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|

&

| $a_j$ | ? | ? | ? | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|

| | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|

# 데스스타 BOJ 11811



# 데스스타 BOJ 11811

- 모든 결과를 이용해 모든 수열에 1이 필수로 필요한 자리를 채우자
- 1로 채우려면 어떻게 해야 할까?

# 데스스타 BOJ 11811

- $a_i$에 결과값을 OR 연산하면 된다

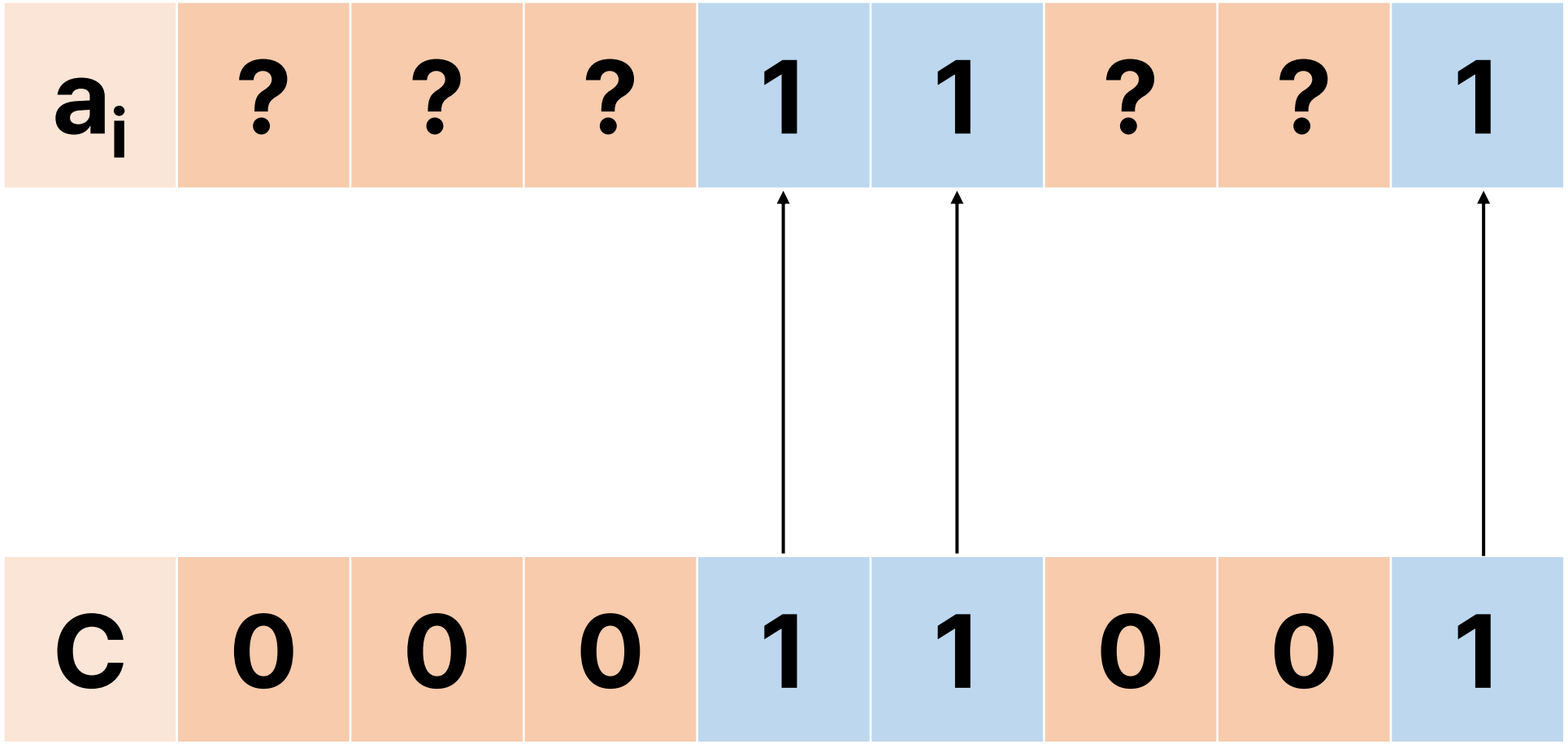
- $a_i$에 결과값을 OR 연산하면 결과값 중 1인 부분은 1로 채워진다
- 결과값 중 0인 부분은 $a_i$의 값으로 남아있는다

# 데스스타 BOJ 11811

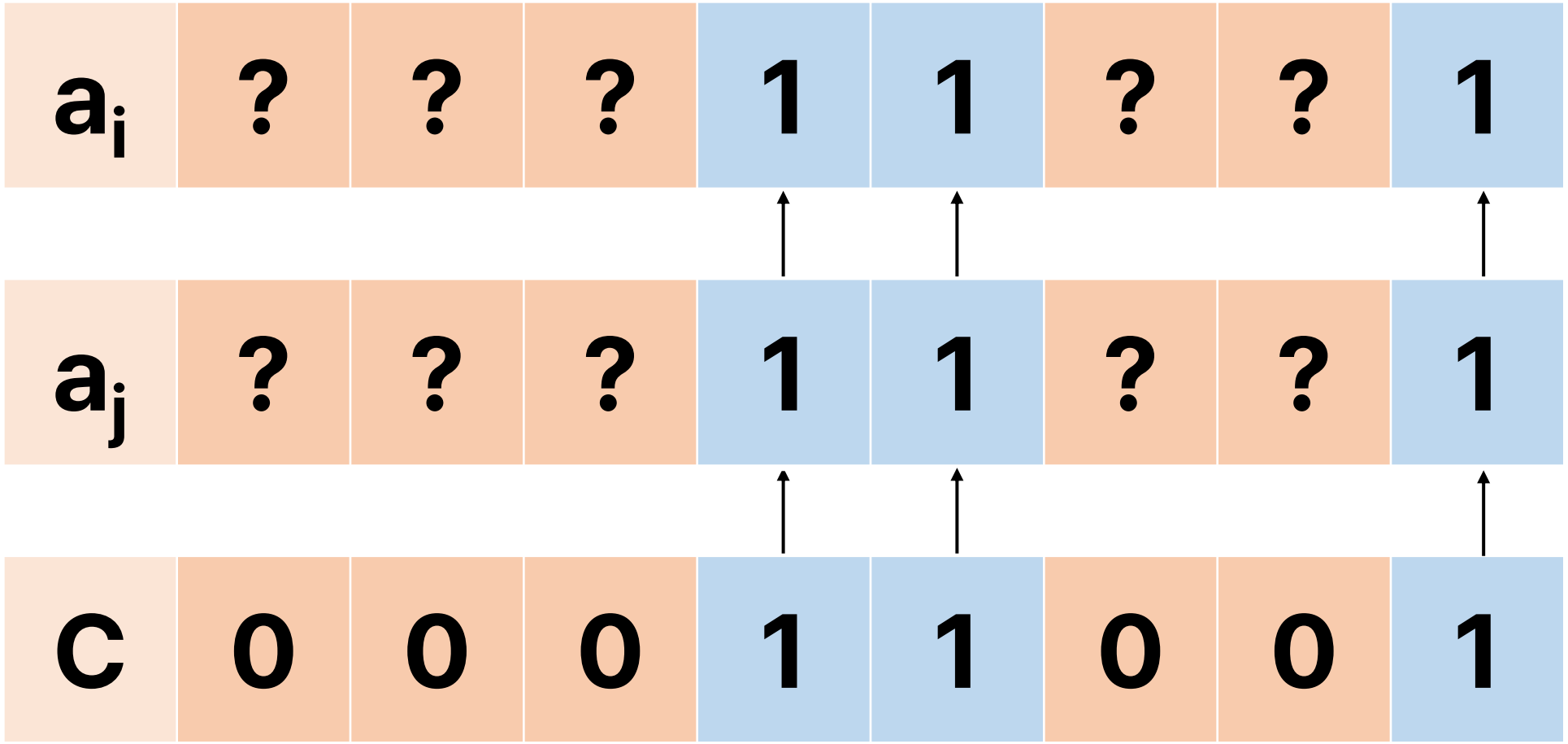
| $a_i$ | ? | ? | ? | ? | ? | ? | ? | ? |
|---|---|---|---|---|---|---|---|---|

| C | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|

# 데스스타 BOJ 11811



# 데스스타 BOJ 11811



# 데스스타 BOJ 11811

- 모든 결과를 이용해 모든 수열에 1이 필수로 필요한 자리를 채우자
- ?에는 0과 1중 어떤 것을 채워야 할까?

| $a_i$ | ? | ? | ? | 1 | 1 | ? | ? | 1 |
|---|---|---|---|---|---|---|---|---|

# 데스스타 BOJ 11811

- ?는 결과가 0이 나온 자리이다

- $a_i$와 $a_j$ 둘 다 해당 자리에 0을 가지고 있거나 둘 중 하나만 1을 가지고 있는다
- 항상 가능한 경우만 주어지므로 둘 다 1을 갖게 하는 입력은 들어오지 않는다

# 데스스타 BOJ 11811

- ex) $a_i$가 다른 수와 AND 연산을 한 결과가 해당 자리에 1이 포함
- $a_j$도 다른 수와 AND 연산을 한 결과가 해당 자리에 1이 포함
- 이 경우 $a_i$와 $a_j$ 둘 다 1을 가져가 하므로 모순이 생긴다(답이 존재할 수 없음)

# 데스스타 BOJ 11811

- 이러한 방식으로 해답의 존재유무를 파악하고 가능한 결과가 존재하는 지 확인하는 알고리즘이 존재
- K-SAT
- 그래프 기반 2-SAT 알고리즘이 존재, 매우 어려움

# 데스스타 BOJ 11811

- 1을 넣으려고 한다면 다른 수들을 전부 살펴보아 해당 자리에 1이 있는 수가 존재하는지 확인해야한다
- 존재하는 경우, 연산 결과가 달라지므로 1을 넣을 수 없다

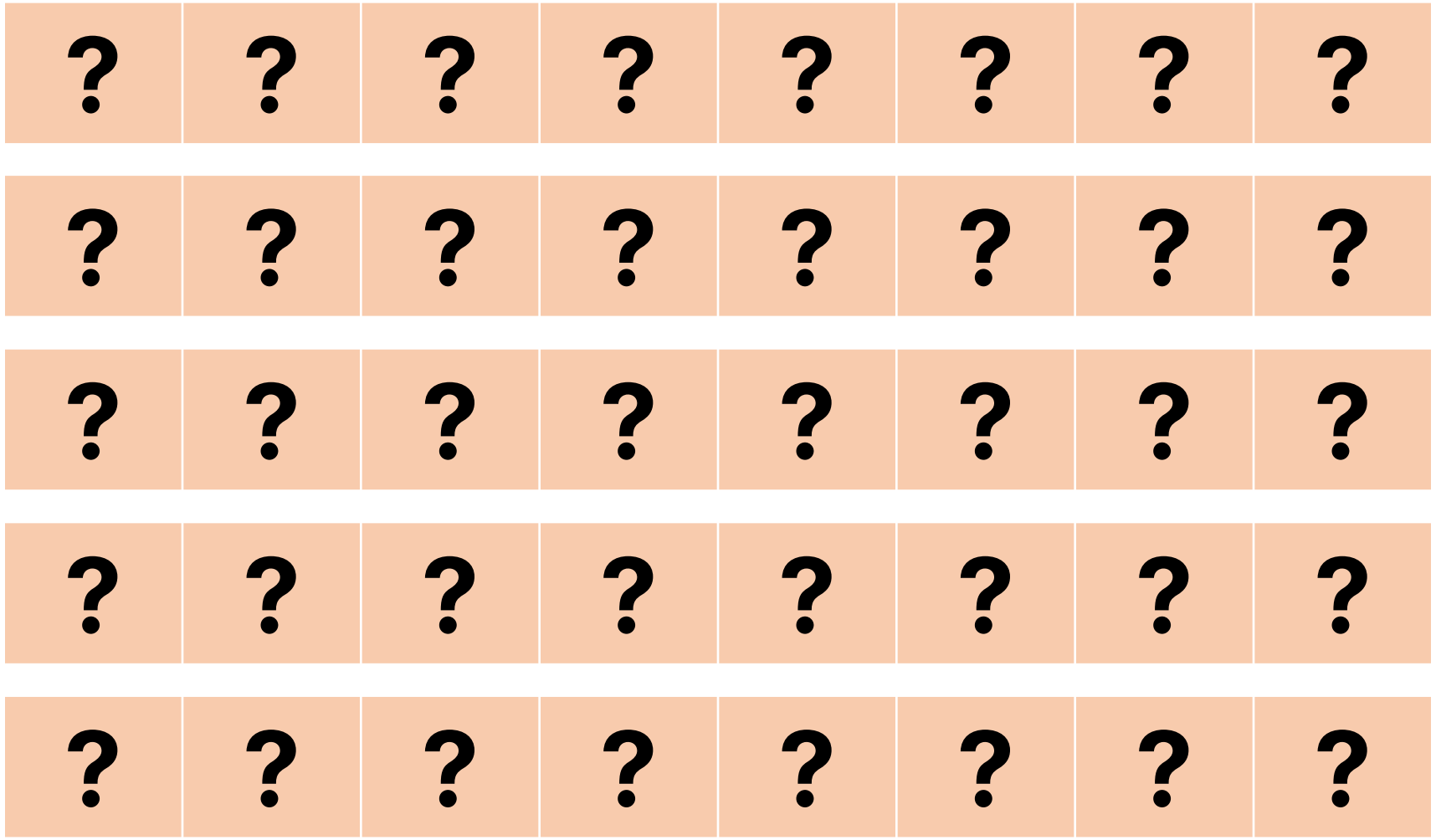
- 모든 경우의 수를 찾아야 한다면 1을 고려해야 하지만 가능한 경우 중 한가지만 얻어내면 되므로 ?자리에는 0을 넣어주도록 하자

# 데스스타 BOJ 11811

- 따라서 종합해보면 0으로 전부 초기화 해두고, 결과값을 OR 연산해 필요한 자리에 1을 넣으면 된다

- 그렇게 나오는 결과는 가능한 경우 중 한 가지이다
- 다른 해답 또한 존재

# 데스스타 BOJ 11811



# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | ? |

# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | ? |

# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |

# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | ? |
| ? | ? | ? | ? | ? | ? | 1 | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |

# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | ? |
| ? | ? | ? | ? | ? | ? | 1 | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | 1 |

# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | ? |
| ? | ? | ? | ? | ? | ? | 1 | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | 1 |

# 데스스타 BOJ 11811

- 남은 자리 중 ?자리는 모든 결과를 확인 했을 때 AND 연산의 결과가 0이 나온 부분이다
- 즉 i번째 숫자와 j번째 숫자를 비교했을 때, {0, 0}, {0, 1}, {1, 0} 3가지 경우 중 한 가지라는 뜻이다
- {1, 1}인 경우, AND 연산의 결과에서 1이 나와서 이미 고정되어 있어야 한다

# 데스스타 BOJ 11811

- i번째 숫자만 집중하자
- ?자리에 0을 넣을 경우와 1을 넣을 경우를 따져보자
- ?자리에 어떠한 비트를 넣었을 때, 입력으로 들어온 AND 연산의 값이 달라지지 않는다면 가능한 답 중 한가지이다

# 데스스타 BOJ 11811

- 0을 넣은 경우, 다른 숫자를 확인할 필요가 없다
- i번째 숫자가 0인 경우, 다른 숫자들과 무관하게 항상 AND한 결과가 0이므로 우리가 의도하는 결과가 나오게 된다

- 따라서 0을 넣는게 간편하므로 0을 넣으면 정답 중 한가지를 간단하게 알아낼 수 있다는 것이다

# 데스스타 BOJ 11811

- 1을 넣은 경우, 다른 숫자 중에서 해당 자리에 1이 존재하는 숫자가 있어서는 안된다

- 5번째 숫자를 가정해보자

# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | ? |
| ? | ? | ? | ? | ? | ? | 1 | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | 1 |

# 데스스타 BOJ 11811

- 파란색 자리에 1이 들어간다고 가정해보자
- 해당 자리는 ?였으므로, 1, 2, 3, 4번째 숫자와 AND한 결과가 항상 0이었다는 것을 의미한다
- 즉, 1, 2, 3, 4번째 숫자는 해당 자리의 비트가 0이어야지 파란색 자리가 1일 수 있다는 것을 의미한다

# 데스스타 BOJ 11811

- 따라서 다른 숫자들이 해당 자리 비트를 0으로 넣고 파란색 자리를 1로 넣으면 가능한 답이 완성된다
- 즉, 우리가 풀이한 결과로는 예제 2번의 결과가 1 2 3 1 3이 나오지만 11이 나올 수 있는 경우 중 한가지이다

# 데스스타 BOJ 11811

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | 1 | ? | ? | 1 |
| ? | ? | ? | ? | 1 | ? | 1 | ? |
| ? | ? | ? | ? | ? | ? | 1 | 1 |
| ? | ? | ? | ? | ? | ? | ? | 1 |
| ? | ? | ? | ? | ? | ? | 1 | 1 |

# 데스스타 BOJ 11811

- 다음과 같이 1번째 수와 2번째 수가 해당 자리에 비트가 1로 고정되었다 가정해보자
- 5번째 수는 ?이므로 0 또는 1 중에서 고를 수 있지만, 다른 모든 수와 AND한 결과가 0이 나와야한다
- 즉, 1번째 수와 AND를 한다면 1과 ?를 AND해서 0이 나와야하므로, ?에는 반드시 0이 와야한다

# 데스스타 BOJ 11811

- 즉 이렇게 1을 넣으려고 할 때는 다른 수 모두가 ?비트인지 확인해야한다

  (지금까지 1로 고정된 적이 없는지 확인해야한다)

- 이렇게 까다로운 경우들이 존재하므로, 좀 더 복잡한 코드가 되므로 간단하게 답을 얻기 위하여 ?에 0을 사용하는 것이다

# Coordinate Compression

# 제일 많은 숫자

- 간단한 예시를 생각해보자
- 1부터 100까지 범위의 숫자가 10만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 어떻게 코딩할까?

# 제일 많은 숫자

- 100까지 개수를 저장할 배열을 만들고 개수를 늘려주며 제일 많은 것을 세주면 된다

```
int num;
int cnt[101];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

# 제일 많은 숫자

- 1부터 2,147,483,647까지 int 양수 범위의 숫자가 10만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 어떻게 코딩할까?

# 제일 많은 숫자

- 과연 가능할까?

```
int num;
int cnt[2147483648];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

# 제일 많은 숫자

- 배열의 크기는 어떨까?
- int이므로 한 칸에 4바이트이다
- 즉, 2,147,483,648칸 * 4바이트 이므로 8,589,934,592바이트이다
- 약 8.5기가의 메모리를 사용해야한다

# 제일 많은 숫자

- 다른 문제로 살펴보자
- 1, 100, 10000 3가지 숫자 10만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 어떻게 코딩할까?

# 제일 많은 숫자

- 앞선 예시를 그대로 사용할 수 있다

```
int num;
int cnt[10001];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

# 제일 많은 숫자

- 하지만 값이 3개만 들어오는데 모든 변수를 사용하지 않는다

```
int num;
int cnt[3];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    if (num == 1)
        cnt[0]++;
    if (num == 100)
        cnt[1]++;
    if (num == 10000)
        cnt[2]++;
}
```

# 제일 많은 숫자

- 어떤 방식을 더 많이 사용하는가?

```
int num;
int cnt[10001];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    cnt[num]++;
}
```

```
int num;
int cnt[3];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    if (num == 1)
        cnt[0]++;
    if (num == 100)
        cnt[1]++;
    if (num == 10000)
        cnt[2]++;
}
```

# 제일 많은 숫자

- 우리는 필요한 만큼의 변수 공간만 만들어 사용한다
- 또한, 비트마스킹처럼 각 배열 인덱스에 의미를 부여할 수 있다

# 제일 많은 숫자

- cnt배열은 각 1, 100, 10000의 개수를 의미한다

```
int num;
int cnt[3];

for (int i = 0; i < 100000; i++) {
    cin >> num;
    if (num == 1)
        cnt[0]++;
    if (num == 100)
        cnt[1]++;
    if (num == 10000)
        cnt[2]++;
}
```

# 제일 많은 숫자

- 1부터 2,147,483,647까지 int 양수 범위의 숫자가 10만개가 입력으로 주어진다
- 이 때, 가장 많은 숫자는 어떤 것인지 출력하시오

- 문제를 다시 살펴보자
- 우리는 2,147,483,647개의 숫자를 위한 공간을 만들 필요가 없다

# 제일 많은 숫자

- 1부터 2,147,483,647까지 int 양수 범위의 숫자가 10만개가 입력으로 주어진다

- 문제에서는 10만개의 숫자가 들어온다고 적혀 있다
- 몇 종류의 숫자가 들어올지는 모르지만 제일 많은 경우, 10만 종류의 숫자가 들어온다
- 따라서 우리는 10만개의 숫자를 위한 공간만 만들어 주면 된다

# 좌표 압축

- 입력으로 주어지는 서로 다른 두 수 사이에 존재하는 사용하지 않는 수들을 우리는 존재하지 않는다 가정하고 사용한다
- 1, 100, 10000이 들어올 때 이 3가지 숫자를 각각 0, 1, 2라고 생각하고 사용하지 않는 수들을 없다고 생각하는 것이다
- 사용하지 않는 수를 없애고 값들을 압축하는 방식, 좌표 압축이라고 부른다

# 좌표 압축

- 좌표 압축은 어디에 쓰일까?

- 모든 문제에 쓰일 수 있다
- 단순히 모든 문제의 입력을 귀찮게 만들 수 있다
- 문제의 알고리즘을 알아도, 입력 그대로 사용한다면 메모리 초과를 만들게 하는 문제

# 좌표 압축

- 좌표 압축은 어디에 쓰일까?

- 모든 문제에 쓰일 수 있다
- 단순히 모든 문제의 입력을 귀찮게 만들 수 있다
- 문제의 알고리즘을 알아도, 입력 그대로 사용한다면 메모리 초과를 만들게 하는 문제

# 좌표 압축

- 앞선 문제에서도 숫자 10만개를 주어질 때 제일 많은 수를 세는 프로그램은 크게 어렵지 않다
- 하지만 입력의 범위에 따라서 기존에 풀 수 있던 문제를 못 풀 수 있다

# 좌표 압축

- 그렇다면 어떻게 하면 될까?

- 우리는 들어오는 수들을 모두 기억하고 중복된 수가 존재하는지 확인해야한다
- 그 이후에 각각의 수들에게 연속되는(1, 2, 3, 4,...) 번호를 부여해야 한다

# 좌표 압축

- 45 1 60 20 78 30 1 20 20 30
- 10개의 수가 입력으로 들어온다고 가정하자
- 중복을 제거하는 경우 45 1 60 20 78 30 6개의 숫자가 남는다

- 우선 들어왔던 수들을 기억하고 있어야 한다
- 기억하고 있어야 중복된 수가 들어오는지를 알 수 있다

# 좌표 압축

- 값들이 들어올 때, 존재하는지 확인하려면 어떻게 해야 할까?

- 들어왔던 모든 수를 탐색하며 중복인지를 확인해야한다
- 들어왔는지 확인하는 배열을 사용하려면 마찬가지로 수의 범위만큼 배열의 칸이 필요하므로 불가능하다

# 좌표 압축

- 지금까지 들어온 수 들을 결국 모두 확인해야한다
- 모든 수 N개에 대하여 K번째 수는 K-1개의 숫자를 확인하므로 $O(N^2)$의 시간복잡도를 보인다(비효율적)

- 매번 확인을 하지 말고 마지막에 한번에 확인을 하자

# 좌표 압축

- 모든 수를 배열에 넣는다
- 모든 수를 정렬하면 같은 수들은 묶인다
- 정렬의 시간복잡도는 $O(\log N)$이므로 훨씬 빠르다

# 좌표 압축

- 또한 정렬하는 이유는 또 존재한다
- 좌표 압축을 하는 경우 무작위로 값을 부여하는 경우, 수의 대소관계가 직관적으로 보이지 않는다
- 입력으로 주어지는 수를 오름차순에 맞추어 좌표 압축 값을 부여하면 수의 **대소관계를 유지**하면서 좌표압축을 할 수 있다

# 좌표 압축

- 기존의 수를 좌표 압축된 수로 바꾸어 줘야 한다
- ex) 45는 3(0 base) 또는 4(1 base)로 바꾸어 줘야 한다

- 따라서 우리는 기존의 수가 몇 번째 수인지 찾아야한다

| 1 | 20 | 30 | 45 | 60 | 78 |
|---|---|---|---|---|---|

# 좌표 압축

- 두가지 방법이 존재한다. 주로 이분 탐색을 사용한다

- 브루트포스 $O(N)$
- 이분 탐색 $O(\log N)$

# 좌표 압축

- 인덱스를 나타내는 두가지 방법

- 각각 점수가 100 95 90 90 85인 5명이 있다고 해보자
- 100점은 1등, 95점은 2등 90점은 3등이다
- 85점은 4등인가 5등인가?

# 좌표 압축

- 점수를 중복 없이 따졌을 때는 100 95 90 85점 순서이므로 85점은 4등이다
- 즉, 인원의 수의 상관없이 인덱스를 빽빽하게 채우려는 경우 사용한다
- 또는, 점수 기준으로 몇 번째인지 알기 위해 사용한다

# 좌표 압축

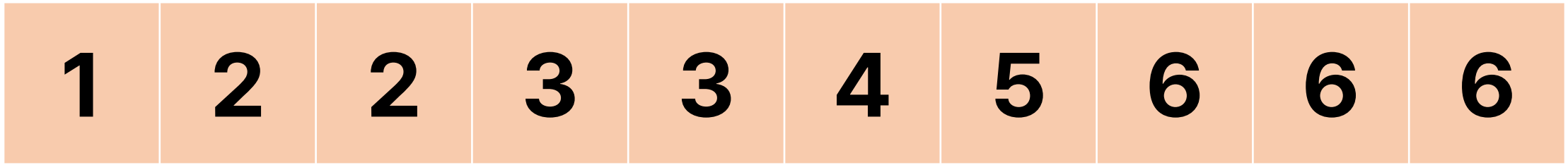
- 중복된 수를 제거하고 몇 번째 수인지 찾아야한다
- 배열에 넣을 때, 이 전에 들어온 수를 찾는 법은 모든 수를 다 확인해야 하므로 비효율적이다
- 벡터에서 unique라는 함수를 사용하면 된다

# 중복 제거

- 정렬된 배열에서 앞에서부터 값 중복되지 않은 값으로 채워주는 unique라는 함수가 존재한다
- algorithm헤더가 필요하다
- 대부분 벡터에서 원소를 제거하는 erase와 같이 자주 사용하기 때문에 벡터를 이용하는 경우가 많다

# 중복 제거

- 1 2 2 3 3 4 5 6 6 6이라는 정렬된 벡터가 존재한다 해보자
- unique는 값의 순서를 바꿔주지 않는다
- 즉, 기존 데이터의 보존을 하지 않는다



# 중복 제거

- 앞에서부터 중복되지 않게 수를 채워준다
- 앞에서부터 1 2 3 4 5 6을 채워나간다

| 1 | 2 | 2 | 3 | 3 | 4 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

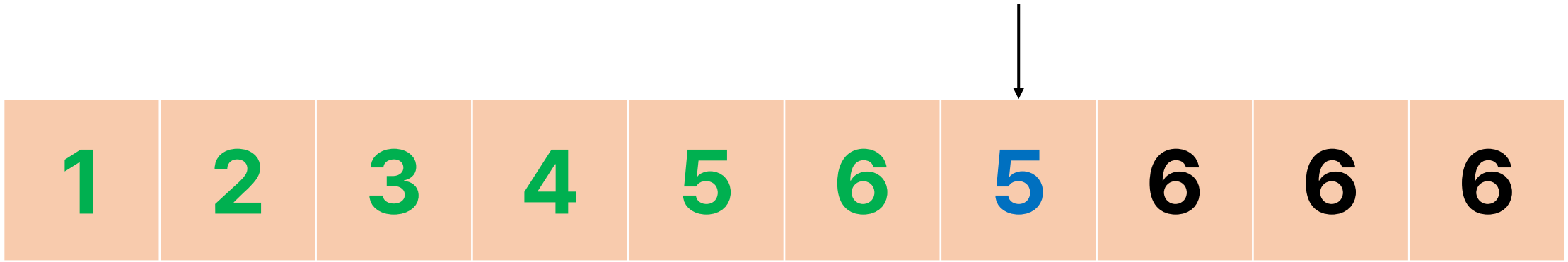
# 중복 제거

- 실행 후 결과

| 1 | 2 | 3 | 4 | 5 | 6 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

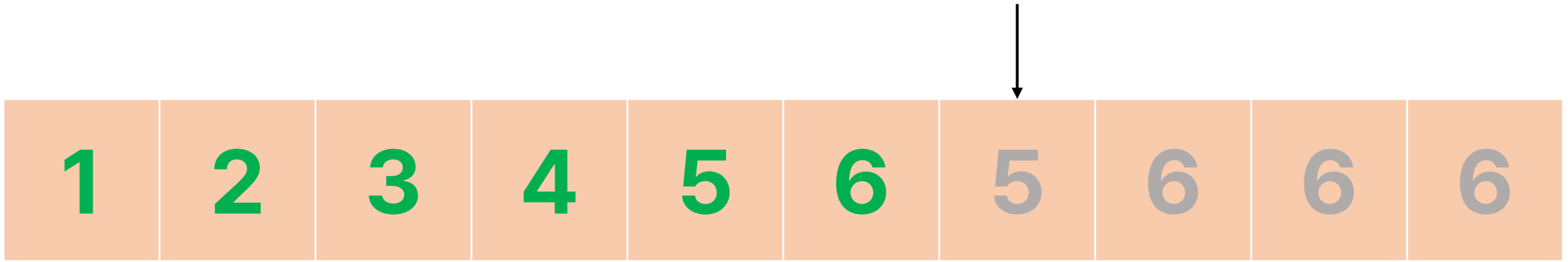
# 중복 제거

- unique는 실행 후 중복되지 않은 수의 뒤 첫번째 숫자를 알려준다
- 해당 예시에서는 파란색 5를 가르킨다



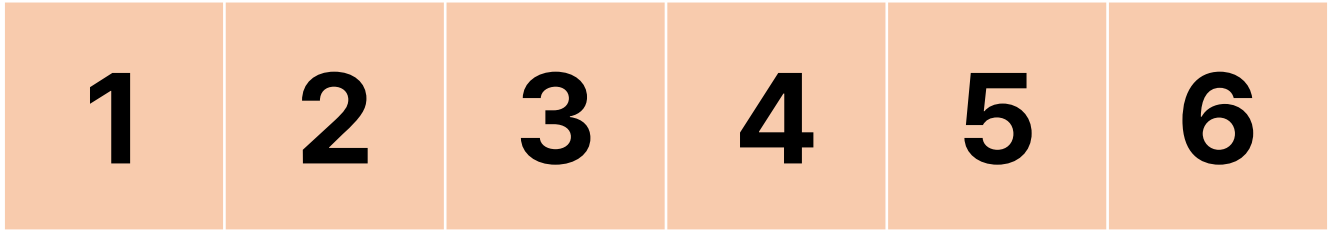
# 중복 제거

- 해당 리턴값을 이용해 벡터에서 erase를 사용한다
- erase를 이용해 해당 위치부터 끝까지 벡터에서 제거한다



# 중복 제거

- erase까지 수행한 뒤에는 중복되지 않은 수만 남아있는다



# 중복 제거

```
vector<int> v = {1, 2, 2, 3, 3, 4, 5, 6, 6, 6};

sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

# 좌표 압축

- 중복된 수를 제거할 필요없이 정렬된 배열에서 몇 등인지만 알면 된다
- 수가 우리가 다룰 수 있는 개수만큼만 들어오기 때문에 중복을 제거하지 않아도 메모리가 충분하다
- 중복된 수가 없이 들어오는 경우 모든 배열의 인덱스를 사용하지만 그렇지 않은 경우, 메모리 낭비가 존재한다

# 이분 탐색

- 1 ~ 100 사이의 수를 찾는다고 해보자
- 어떤 수 K에 대하여 그 수보다 작다고 하는 경우, K보다 큰 수는 모두 정답이 아닌 것을 알 수 있다
- 어떠한 값에 대해서 문의를 했을 때, 그 결과로 특정 부분에도 동일한 결과를 적용하는 것
- 이것을 중간 지점에 계속 문의를 날리는 방식을 이분 탐색이라고 한다

# 이분 탐색

- 우리가 찾는 범위가 지정이 되어 있다
- 이것을 left~right로 부른다
- 우리가 찾는 범위가 right < left인 경우, 찾는 범위에 해당되는 값이 존재하지 않는다

# 이분 탐색

- 이분 탐색을 이용한 함수가 2개 존재한다
- Upper Bound: 찾으려는 값보다 큰 숫자의 위치를 알려준다
- Lower Bound: 찾으려는 값과 같거나 큰 숫자의 위치를 알려준다

# 이분 탐색

- 우리가 찾으려는 수는 이미 기존에 배열에 있는 값을 찾으므로 반드시 정렬된 배열에 존재한다
- 즉, 기본적인 탐색은 존재하는지, 존재하지 않는지를 판단해야 한다. 하지만 그럴 필요가 없다
- Lower Bound를 사용하면 쉽게 찾을 수 있다

# 이분 탐색

- 우리가 찾으려는 수는 이미 기존에 배열에 있는 값을 찾으므로 반드시 정렬된 배열에 존재한다
- 즉, 기본적인 탐색은 존재하는지, 존재하지 않는지를 판단해야 한다. 하지만 그럴 필요가 없다
- Lower Bound를 사용하면 쉽게 찾을 수 있다

# 이분 탐색

- lower_bound는 어디에 있는지 포인터로 알려준다

```
vector<int> v = {1, 2, 2, 3, 3, 4, 5, 6, 6, 6};
sort(v.begin(), v.end());
lower_bound(v.begin(), v.end(), 1);
```

# 이분 탐색

- 인덱스를 확인하기 위해서는 배열의 시작 포인터를 빼주면 된다

```
vector<int> v = {1, 2, 2, 3, 3, 4, 5, 6, 6, 6};
sort(v.begin(), v.end());
lower_bound(v.begin(), v.end(), 1) - v.begin();
```

# 이분 탐색

- Upper Bound와 Lower Bound를 유용하게 사용하는 방법

- 수의 개수를 셀 수 있다

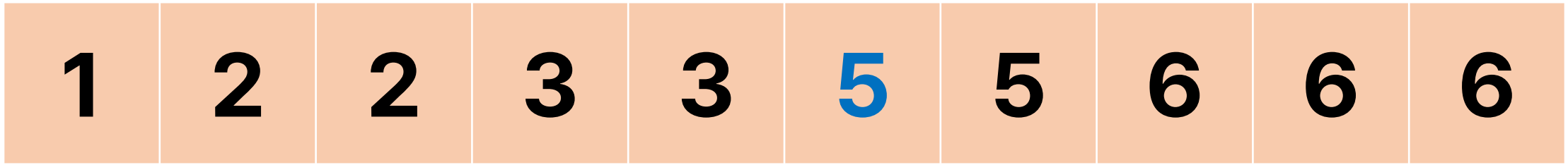
# 이분 탐색

- 수가 존재하지 않는 경우, 4를 찾는다고 가정하자
- 4가 존재하지 않으므로 Upper Bound와 Lower Bound 둘 다 4보다 큰 수를 찾게 된다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색

- Upper Bound와 Lower Bound 둘 다 파란색 5를 찾게 된다
- 둘의 위치를 빼면, 개수가 0이 나옴을 알 수 있다



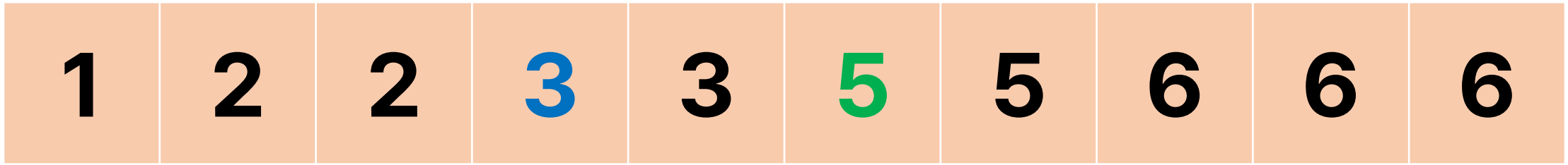
# 이분 탐색

- 수가 존재하는 경우, 3을 찾는다고 가정하자
- Upper Bound는 3보다 큰 수를 찾으며, Lower Bound 3을 찾게 된다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색

- Upper Bound는 초록색 5를 가르킨다
- Lower Bound는 파란색 3을 가르킨다
- 둘의 차는 2개이며, 3이 2개임을 알 수 있다



# Upper Bound를 이용한 최빈값 찾기

- 지금까지 사용한 방식은 각 수의 인덱스를 우리가 사용할 배열의 인덱스로 바꾸는 방법이었다
- 하지만 Upper Bound를 배웠으니 다른 방법을 사용해보자

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색을 이용한 최빈값 찾기

- 우선 정렬이 되어 있어야 한다
- 우리가 왼쪽에서부터 수를 찾기 시작하면 현재 찾는 수가 몇 개 인지 모르지만, 제일 먼저 나온 수는 그 중에서 제일 왼쪽에 있는 수 이다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색을 이용한 최빈값 찾기

- 파란색 1은 제일 왼쪽에 있는 1이다
- 제일 먼저 나오는 3은 제일 왼쪽에 있는 3이다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색을 이용한 최빈값 찾기

- 즉, 우리는 Lower Bound를 따로 찾지 않아도 된다
- 그리고 해당 값은 나보다 작은 수의 Upper Bound이다

- 하나씩 해보자

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색을 이용한 최빈값 찾기

- 1의 Lower Bound는 제일 왼쪽에 있는 1이므로 우리는 이것을 이미 알고 있다
- 따라서 우리는 1의 Upper Bound를 찾아내면 1의 개수를 알 수 있다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색을 이용한 최빈값 찾기

- 제일 첫 숫자는 제일 작은 숫자이다
- 1의 개수를 세기 위해서는 1의 Upper Bound와 Lower Bound를 찾아야 한다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색을 이용한 최빈값 찾기

- 1의 Upper Bound를 이용해 우리는 1의 개수를 알아냈다
- 그 다음 개수를 찾을 숫자로 이동하려면 나보다 크면서 제일 작은 숫자의 위치로 이동해야 한다
- 즉, Upper Bound의 위치로 이동하면 된다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

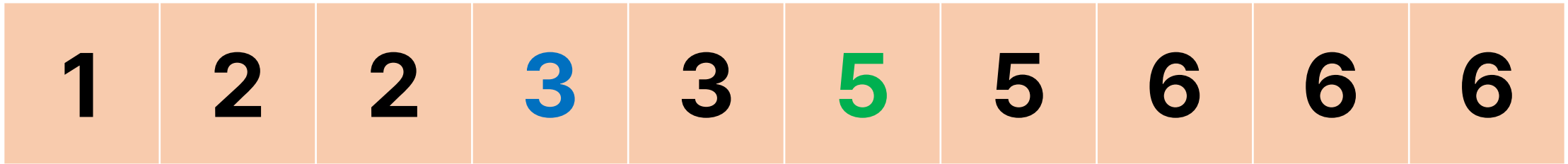
# 이분 탐색을 이용한 최빈값 찾기

- 2로 이동한 뒤에는 똑같이 반복하며 수의 개수를 알아낼 수 있다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

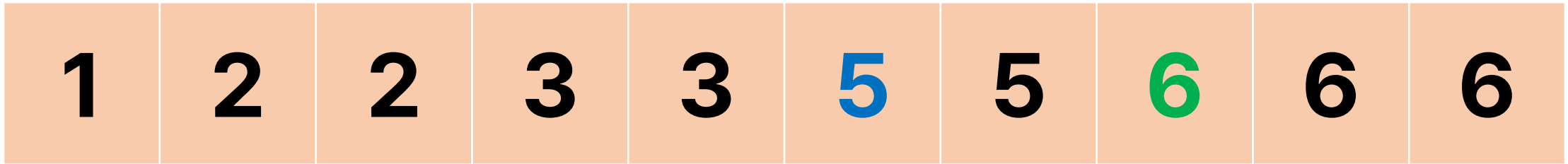
# 이분 탐색을 이용한 최빈값 찾기

- 반복



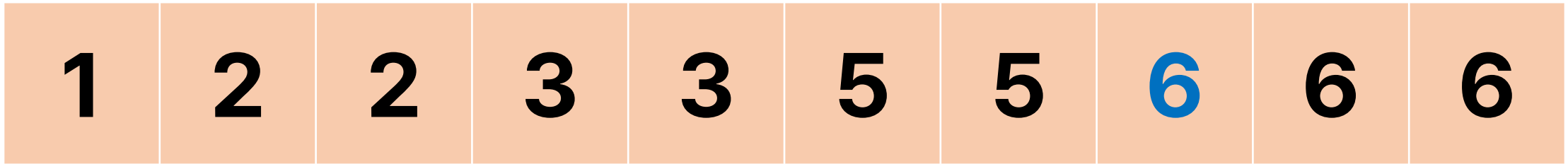
# 이분 탐색을 이용한 최빈값 찾기

- 반복



# 이분 탐색을 이용한 최빈값 찾기

- 반복



# 이분 탐색을 이용한 최빈값 찾기

- 이후에 현재 위치가 배열의 끝이라면 모든 수를 탐색했으므로 종료하면 된다

| 1 | 2 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 6 |
|---|---|---|---|---|---|---|---|---|---|

# 이분 탐색을 이용한 최빈값 찾기

```
vector<int> v;
// insert

sort(v.begin(), v.end());
auto cur = v.begin();
while (cur != v.end()) {
    auto next = upper_bound(v.begin(), v.end(), *cur);
    int cnt = next - cur;
    cur = next;
}
```

# Set/Map

# Set/Map

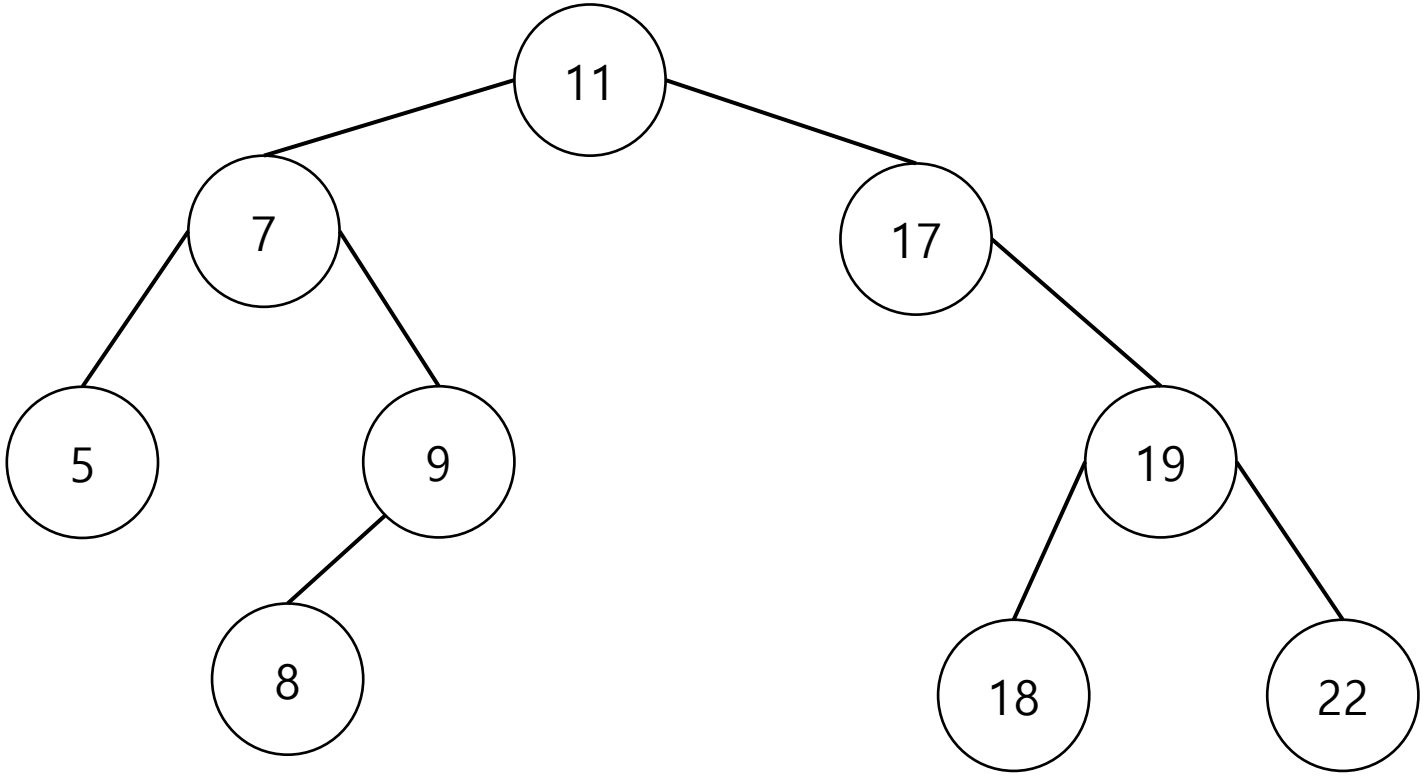
- 배열에 단순히 삽입 삭제하는 경우, 중복된 값이 존재하는지 확인하기 어렵다
- 지금까지 배운 내용도 모든 데이터가 들어온 경우에 중복을 확인하는 경우이다
- 실시간으로 삽입과 쿼리(확인) 과정이 일어나면, 배열 같은 경우에는 다시 정렬하는 과정이 필요하다
- 즉, 배열을 사용하는 것은 매우 느리다

# Set/Map

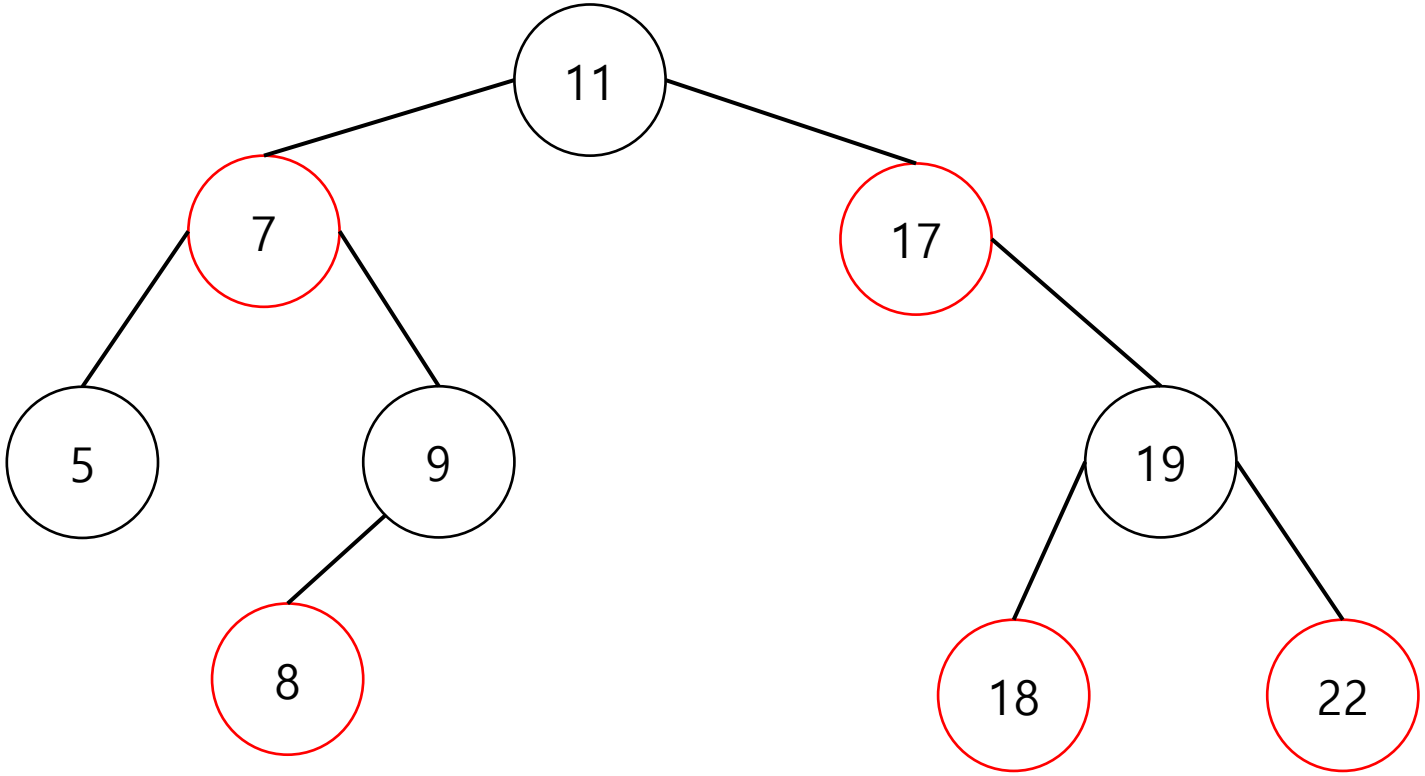
- 간단하게 설명했던 이진 트리의 개념으로, 배열보다 빠른 속도로 중복된 값이 존재하지 않도록 삽입과 삭제를 하는 자료구조이다
- C++에는 Red/Black tree라는 이진 트리로 구현되어 있다

- 중요한 점은 Set과 Map에 삽입된 데이터는 중복이 없으며, 정렬된 구조를 유지한다는 점이다

# Set/Map



# Set/Map



# Set/Map

- Set은 데이터 자체가 찾으려는 키인 자료구조이다
- Map은 검색하는 단어와 데이터가 존재하는 자료구조이다

# Set/Map

- Set과 Map은 인덱스를 지원하지 않는다
- 즉, Set에 1, 2, 3을 insert 했을 때, 2가 몇 번째 수인지 찾을 수 없다는 것이다
- 첫 번째 수부터 보면서 2가 몇 번째 인지는 확인할 수 있지만, 인덱스 자체를 구하라는 연산은 불가능하다

# STL Set

- #include <set>
- set<자료형> 변수명

# STL Set

- set.insert(k): set에 k라는 값이 존재하지 않으면 삽입한다
- set.find(k): set에서 k라는 값을 찾는다. 존재하지 않는 경우 set.end()를 돌려준다
- set.erase(k): set에서 k라는 값을 지운다
- set.clear(): set의 모든 원소를 지운다

# STL Map

- #include <map>
- map<key 자료형, value 자료형> 변수명

# STL Map

- map.insert(key, value): map에 k라는 키를 가진 값이 존재하지 않으면 삽입한다
- map.find(key): map에서 k라는 값을 찾는다. 존재하지 않는 경우 map.end()를 돌려준다
- map.erase(key): map에서 k라는 키를 가진 값을 지운다
- map.clear(): map의 모든 원소를 지운다

# 문제

- 좌표 압축 BOJ 18870
- N차원 여행 BOJ 12867
- 개수 세기 BOJ 10807
- 통계학 BOJ 2108