OT

25.03.24 (월) 오후 5시 ~ 7시

목차

- 시간 복잡도
- 정렬
- 투 포인터
- 이분 탐색
- 매개변수 탐색
- 분할 정복

시간 복잡도

시간 복잡도 | 01 시간 복잡도의 개념

- 시간 복잡도는 **입력 크기**가 커질수록 알고리즘의 **실행 시간**이 어떻게 변하는지를 나타내는 개념이다.
- 여러 표기법이 있는데, **PS에서는 주로 빅오 표기법을 사용**한다. 최악의 경우를 나타내 주기 때문이다.

시간 복잡도 | 02 빅오(Big-O) 표기법

- 빅오 표기법은 **최악의 경우 수행 시간을 나타내는 상한 (점근적 상한)** 을 의미한다.
- 주어진 시간 복잡도를 빅오 표기법으로 나타내려면, **가장 영향력이 큰 항**만 남기면 된다.

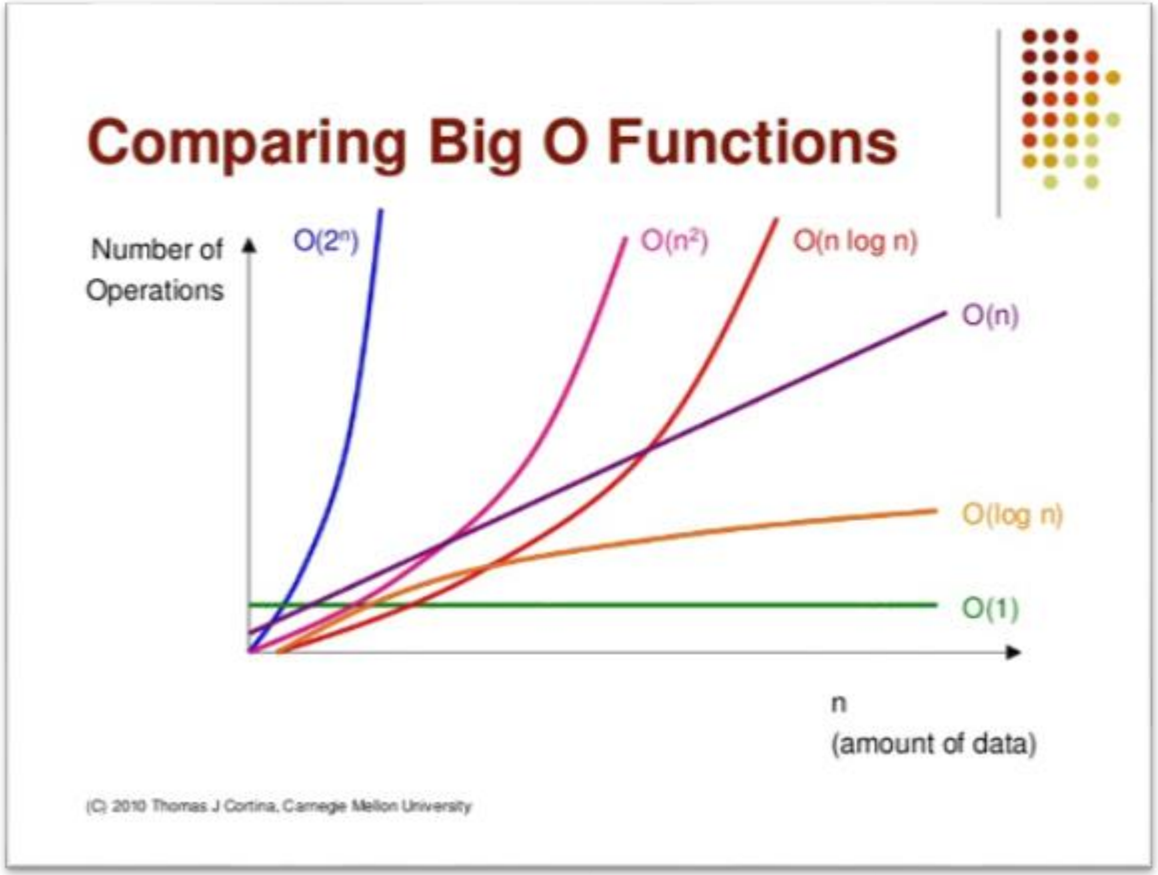
ex) $f(n) = 2^n + 10n$ 의 시간복잡도는 $O(2^n)$ 이다.

n	2^n	10n	영향력
1	2	10	10n이 크다
3	8	30	10n이 크다
20	1,048,576	200	2^n이 크다
30	1,073,741,824	300	2^n이 크다

시간 복잡도 | 02 빅오(Big-O) 표기법

주어진 시간 복잡도를 빅오 표기법으로 나타내는 연습을 해보자.

- $a(n) = n \log n + 10n \ \rightarrow \mathrm{O(n \log n)}$
- $\mathrm{b}(n) = \mathrm{n^n + n!} \ \rightarrow \mathrm{O}(n^n)$
- $c(n) = \mathrm{\log n + n} \rightarrow \mathrm{O}(n)$
- $d(n) = 2n^2 + 10n \ \rightarrow \mathrm{O(n^2)}$
- $e(n) = 2^n + 10n \ \rightarrow \mathrm{O(2^n)}$
- $f(n) = 2n^3 + 10n \ \rightarrow \mathrm{O(n^3)}$
- $g(n) = 10 \log n + 4 \ \rightarrow \mathrm{O}(\log n)$



시간 복잡도 | 02 빅오(Big-O) 표기법

- 알고리즘 문제에서는 입력 크기를 주어주고 제한 시간을 준다.
  이를 통해 **어느 정도의 시간 복잡도를 가진 알고리즘을 생각해야 하는지 예상**이 가능하다.

컴퓨터는 1초에 1억번(10^8) 동작한다고 생각하면,

- "N이 5,000이면 O(N^2)까지는 가능하겠구나. "
- "N이 200,000이면 O(N log N) 정도의 알고리즘을 짜야겠구나."

라는 생각을 할 수 있다.

시간 복잡도 | 03 예시

- 배열의 원소 접근
  - 배열에서 k 번째 원소의 메모리 주소는 다음과 같이 계산한다.

    (배열의 시작 주소) + k * (자료형의 크기)
  - 한 번의 산술 연산만 필요하므로 시간 복잡도는 O(1)이다.
- 이중 for문
  - 바깥 루프는 N번 실행되고, 바깥 루프가 한 번 실행될 때마다 안쪽 루프는 다시 N번 실행된다.
  - 총 실행 횟수 $T(N) = N * N = N^2$이다.
  - 따라서, 시간 복잡도는 O(N^2)이다.

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        cout << i << ", " << j << endl;
    }
}
```

시간 복잡도 | 03 예시

- 주어진 배열에서 최대값, 최소값 찾기
  - 연산 : 두 원소의 값 비교
  - 수행 횟수 : 최대 N – 1회
  - 시간 복잡도 : $T(N) = N - 1$
  - 빅오 표기법을 활용하면 O(N)이다.
- 주어진 배열에서 특정한 값 X가 있는지 탐색하기
  - 연산 : 두 원소의 값 비교
  - 수행 횟수 : 최대 N회
  - 시간 복잡도 : $\mathrm{H}(N) = N$
  - 빅오 표기법으로 나타내면 O(N)이다.

정렬

정렬 | 01 개요

- 정렬은 주어진 데이터를 특정한 순서로 재배열하는 것을 의미한다.

  오름차순 정렬 : a < b 이면 a가 b보다 앞에 오도록 나열

  내림차순 정렬 : a > b 이면 b가 a보다 앞에 오도록 나열

- Stable Sort vs Unstable Sort
  - stable sort : 동일한 값을 가진 원소들의 상대적인 순서가 유지되는 정렬. C++ <algorithm> stable_sort
  - unstable sort : 동일한 값을 가진 원소들의 상대적인 순서가 유지되지 않는 정렬. C++ <algorithm> sort

- O(N^2) 정렬 알고리즘 : 버블 정렬, 선택 정렬, 삽입 정렬
- O(N log N) 정렬 알고리즘 : 병합 정렬, 퀵 정렬, 힙 정렬

정렬 | 02 $O(N^2)$ 정렬 알고리즘 : 버블 정렬

- 버블 정렬

  가장 단순한 정렬 알고리즘이다.

  인접한 두 원소를 비교해서 정렬하는 방식으로 작동한다.

```cpp
void bubble_sort(vector<int> &arr){
    for(int i = 0; i < arr.size() - 1; i++){
        for(int j = 0; j < arr.size() - i - 1; j++){
            if(arr[j] > arr[j + 1]){
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

- 시간 복잡도 분석

  벡터 arr의 크기를 n이라 하자.

  바깥 루프는 n - 1번 반복한다.

  안쪽 루프는 n – 1, n – 2, …, 1번 반복하며 총 비교 횟수는 (n – 1) + (n – 2) + … + 1 = n * (n – 1) / 2 이다.

  O(n^2)임을 알 수 있다.

정렬 | 03 C++ <algorithm> std::sort

- 함수 원형

```
void sort(RandomIt first, RandomIt last, Compare comp);
```

- 비교함수의 조건 (strict weak ordering)

Strict Weak Ordering은 std::sort와 같은 정렬 알고리즘에서 비교 함수가 만족해야 하는 성질이다.

이를 지키지 못하면 관계를 규정하는 데 모순이 발생해 정렬의 결과가 올바르지 않다.

- 비반사성 : 어떠한 원소도 자기 자신보다 작을 수 없다. x < x는 항상 거짓이여야 한다.

  지켜지지 않으면? 정렬을 수행할 때 동일한 원소끼리도 순서가 정해져야 하는 모순이 발생할 수 있다.

- 비대칭성 : 어떤 두 원소 x, y에 대해, 만약 x < y 라면 y < x 는 항상 거짓이여야 한다.

  지켜지지 않으면? 순환 관계가 발생해서 논리적으로 정렬이 불가능해진다.

- 추이성 : a가 b보다 작고, b가 c보다 작다면, a도 c보다 작아야 한다.

- 비비교성의 추이성 : 어떤 두 원소 a와 b가 서로 비교할 수 없다면, 다른 원소 c에 대해서도 a와 b는 동일하게 동작해야 한다.

정렬 | 04 예제 : 백준 11650번 좌표 정렬하기

- 문제 요약

  2차원 평면 위의 점 N개 가 주어진다.

  x좌표를 기준으로 오름차순 정렬해서 출력하기.

  x좌표가 같다면 y좌표를 기준으로 오름차순 정렬해서 출력하기.

- 시간복잡도

  N = 100,000이므로 O(n log n)의 시간복잡도를 가진 std::sort 함수로 해결 가능!

- 구현

  sort 함수는 기본적으로 오름차순 정렬을 하기 때문에 따로 함수를 작성해주지 않아도 된다.

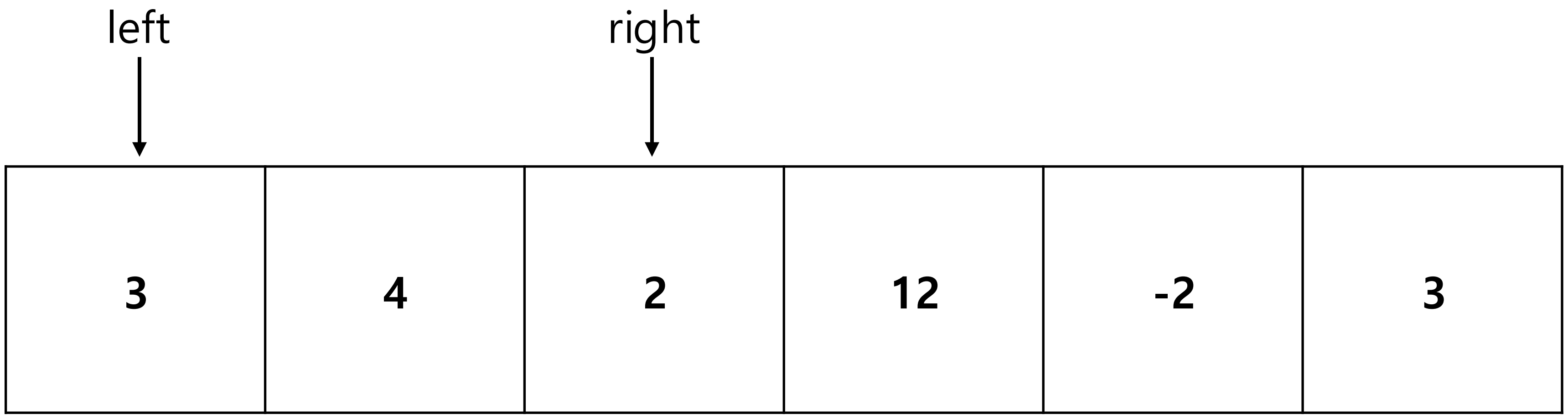
```cpp
#include <bits/stdc++.h>
using namespace std;

int N;
int main(){
    cin >> N;
    vector<pair<int, int>> p(N);
    for(int i = 0; i < N; i++){
        cin >> p[i].first >> p[i].second;
    }
    sort(p.begin(), p.end());
    for(auto k : p){
        cout << k.first << " " << k.second << "\n";
    }
    return 0;
}
```

투 포인터

투 포인터 | 01 개요

- 투 포인터는 1차원 배열상에서 두 개의 포인터를 사용하여 특정 조건을 만족하는 값을 찾거나 부분을 찾아 최적화하는 기법이다.
- 이 기법을 활용하면 O(N^2) 이상의 풀이를 보통 O(n) 또는 O(n log n)으로 최적화할 수 있다.



투 포인터 | 02 예제 : 백준 1806번 부분합

- 문제 요약

  길이 N인 수열 A가 주어진다. (10 <= N < 100,000)

  부분 연속된 구간의 합이 S 이상이 되는 가장 짧은 부분 수열의 길이를 출력한다. (0 < S <= 100,000,000)

  그러한 수열이 존재하지 않는다면 0을 출력한다.

- 시간 제한

  0.5초 -> 컴퓨터가 1초에 할 수 있는 연산은 10^8이므로 5 * 10^7번 이내로 연산을 마쳐야 한다.

  N의 최댓값이 10^5이므로 O(N log N)보다 빠른 알고리즘을 짜야 함을 알 수 있다.

예제 입력 1 복사

```
10 15
5 1 3 5 10 7 4 9 2 8
```

예제 출력 1 복사

```
2
```

투 포인터 | 02 예제 : 백준 1806번 부분합

- 풀이

떠올릴 수 있는 알고리즘은 모든 부분합을 계산해보고, 그 합이 S 이상이 되는 것 중 가장 짧은 것의 길이를 구하는 것이다.

사전 계산이 필요하지만 부분합은 O(1)에 구할 수 있는데 다음을 활용한다.

S[i] = a[0] + a[1] + … + a[i - 1] // S[i]는 a[0] 부터 a[i - 1]까지의 합

이를 이용하면 배열 a의 구간 [L, R]의 합을 O(1)만에 구할 수 있다.

S[R] 값과 S[L - 1] 값을 빼면 구간 [L, R]의 합이 남기 때문이다.

즉, L과 R를 잘 조절해가면 문제의 답을 찾을 수 있다.

투 포인터 기법을 활용할 수 있겠다.

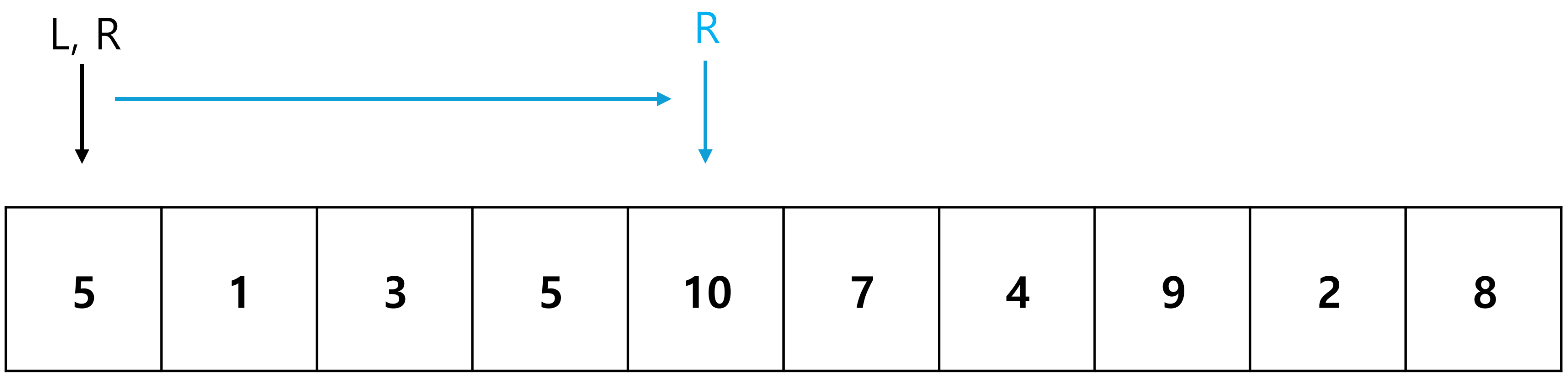
투 포인터 | 02 예제 : 백준 1806번 부분합

- 풀이

L, R을 조절하는 방법?

처음에는 두 개의 포인터 모두 a[0]에서 시작해본다.

부분합이 15가 넘을 때까지 R을 오른쪽으로 옮겨준다.



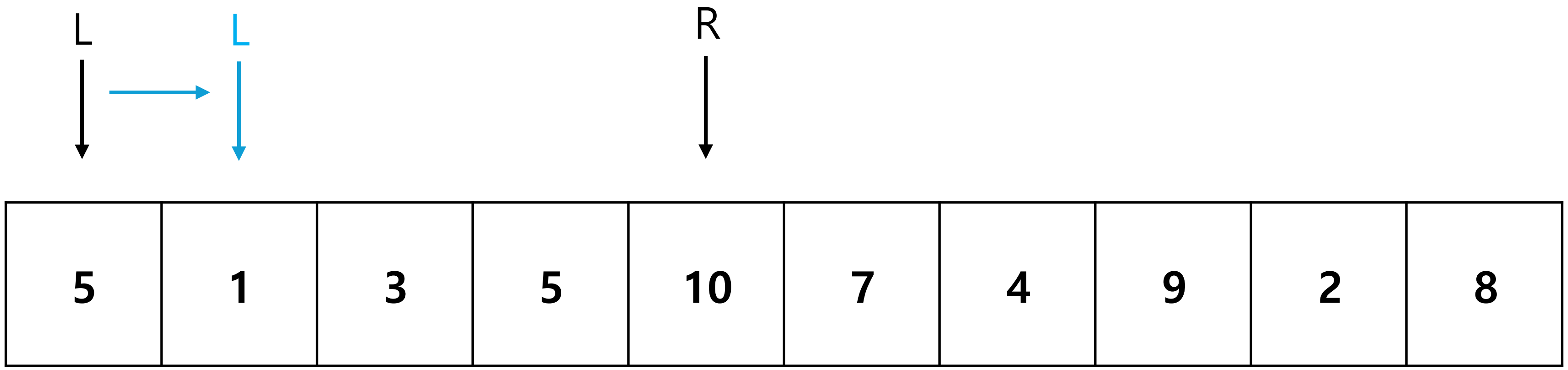
투 포인터 | 02 예제 : 백준 1806번 부분합

- 풀이

R을 더 이상 오른쪽으로 옮길 필요가 없는 이유는 가장 짧은 것의 길이를 구하는 게 문제이기 때문이다.

이제 L을 왼쪽으로 한 칸 옮겨준다. 부분합의 값이 작아지는데, S 이상일 지 미만인지는 확인해야 한다.

여기에서는 부분합이 19로 S 이상이다. L을 한 칸 더 옮겨준다.



투 포인터 | 02 예제 : 백준 1806번 부분합

- 풀이

S 미만이 될 때까지 L를 오른쪽으로 옮겨준다. L을 한 칸 옮기는 과정을 여러 번 반복 수행한 결과와 같다.

옮기는 과정 중에서 구한 최솟값은 2로 구간 [3, 4]의 합이다.

[4, 4]의 부분합은 10으로 S 미만 이므로 R이 S 이상일 때까지 움직인다. 위 과정을 L과 R이 배열의 끝까지 이동할 때 문제의 답을 구할 수 있다.

5	**1**	**3**	**5**	**10**	**7**	**4**	**9**	**2**	**8**

투 포인터 | 02 예제 : 백준 1806번 부분합

- 구현

Line 15번을 보자.

L, R은 0으로 초기화되어 있고 sub_sum은 부분합을 저장하는 배열인데 A[0]로 초기화되어 있다.

Line 16을 보자.

부분합이 S 미만이여야 하고 오른쪽을 가리키는 포인터가 배열의 범위를 초과하면 안 된다.

Line 21을 보자.

sub_sum에서 A[L]을 빼주고 L을 오른쪽으로 한 칸 옮긴다.

```
1   #include <bits/stdc++.h>
2   using namespace std;
3
4   int res = INT32_MAX;
5
6   int A[101010], S, N;
7
8   int main(){
9       ios::sync_with_stdio(false);
10      cin.tie(nullptr);
11
12      cin >> N >> S;
13      for(int i = 0; i < N; i++) cin >> A[i];
14
15      for(int L = 0, R = 0, sub_sum = A[0]; L < N;){
16          while(R < N and sub_sum < S){
17              if(++R != N) sub_sum += A[R];
18          }
19          if(R == N) break; // R이 범위를 벗어날 시 종료
20          res = min(res, R - L + 1);
21          sub_sum -= A[L++];
22      }
23      cout << (res == INT32_MAX ? 0 : res) << "\n";
24  }
```

투 포인터 | 02 예제 : 백준 1806번 부분합

- 시간 복잡도

  두 개의 포인터 L과 R이 한 번씩 움직이게 된다.

  전체적으로 배열의 각 원소가 **최대** 두 번씩만 처리되었다는 의미다.

  작성한 알고리즘의 시간 복잡도는 O(N) + O(N) = O(N)이다.

  모든 부분합들을 구하는 브루트 포스 방식으로 풀게 되면 O(N^2) 이상인데, 투 포인터 기법을 통해 O(N)으로 최적화했다.

투 포인터 | 03 요약

- 핵심
  1. 하나의 배열을 두 개의 포인터 L, R을 사용하여 탐색함.
  2. 특정 조건을 만족할 때까지 포인터를 조정하면서 최적의 결과를 찾음.

- 이동 방식
  - 양 끝에서 시작하여 중앙으로 수렴하는 방식(L = 0, R = n – 1) ex) 백준 2470번 두 용액
  - 한쪽 포인터는 유지하고 다른 포인터를 이동시키는 방식 (방금 본 문제)
  - 두 개의 배열을 비교하는 방식 (나중에 살펴볼 병합 정렬에서의 merge 함수)

이분 탐색

이분탐색 | 01 개요

- 이분 탐색은 **정렬된 배열**에서 특정한 값을 빠르게 찾는 알고리즘이다.
- 기본 아이디어는 탐색 범위를 절반씩 줄여가며 목표 값을 찾는 것으로 O(log N)의 시간복잡도를 가진다.

이분탐색 | 01 개요

- Up & Down 게임으로 이분 탐색이 어떤 느낌인지 알아봅시다.

👩‍🏫 **강사:**

- “여러분, 우리가 **1부터 100 사이의 숫자** 중 하나를 맞히는 **업다운 게임**을 한다고 생각해봅시다.
- 제가 1부터 100 사이의 숫자 중 하나를 정해놓을 테니까, 여러분이 맞혀보세요!”

👩‍🎓 **학생:**

- “음... **50**인가요?”

이분탐색 | 01 개요

🧑‍🏫 **강사:**

- "틀렸어요! 제가 정한 숫자가 **50보다 작아요**. DOWN!"

🧑‍🎓 **학생:**

- "오케이, 그럼 1부터 50 사이에서 찾아야겠네요. 이번엔 **25**!"

🧑‍🏫 **강사:**

- "틀렸어요! 제가 정한 숫자가 **25보다 커요**. UP!"

🧑‍🎓 **학생:**

- "그럼 25부터 50 사이에서 찾아야 하니까... **37**!"

🧑‍🏫 **강사:**

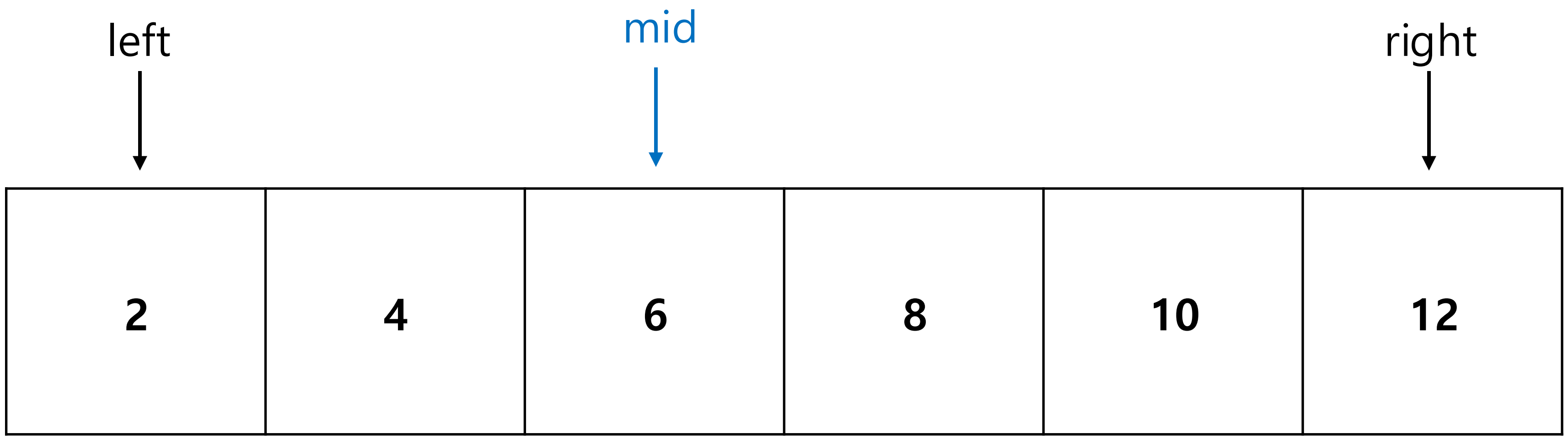
- "맞아요! 정답은 37이었습니다!" 🎉

이분탐색 | 01 개요

예시를 통해 이분탐색의 동작 과정을 알아보겠습니다.

배열 [2, 4, 6, 8, 10, 12]에서 10을 찾는 이분탐색을 진행해보자.

처음의 탐색 범위는 [0, 5]이다. 두 범위의 중간 지점은 (0 + 5) / 2 = 2이다.

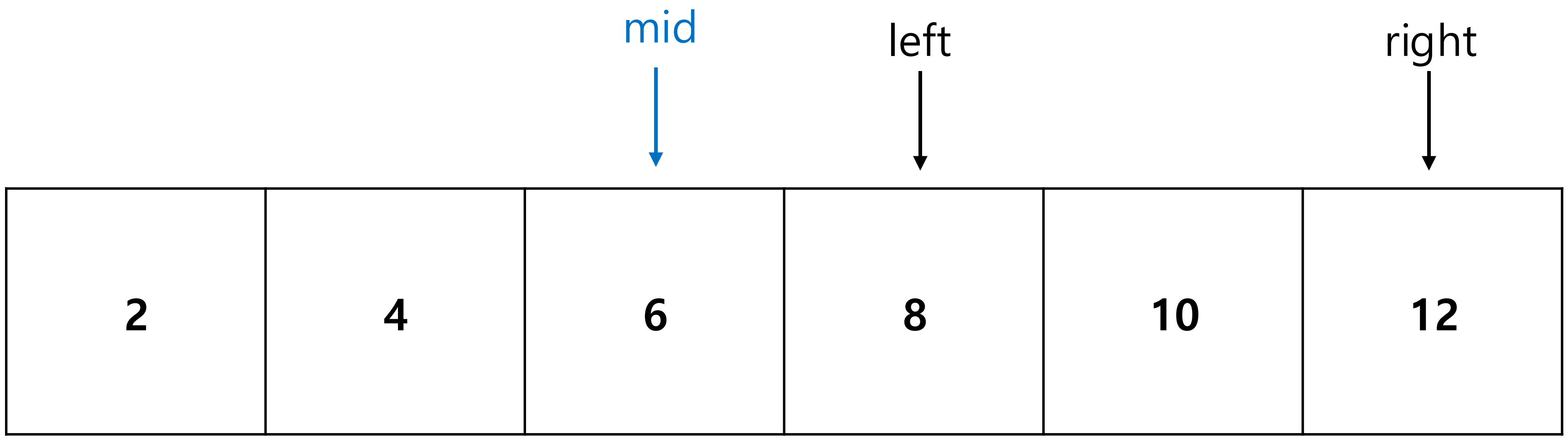


이분탐색 | 01 개요

mid에 위치한 값이 찾고자 하는 값인 10보다 작다.

left를 mid + 1로 바꾼다.

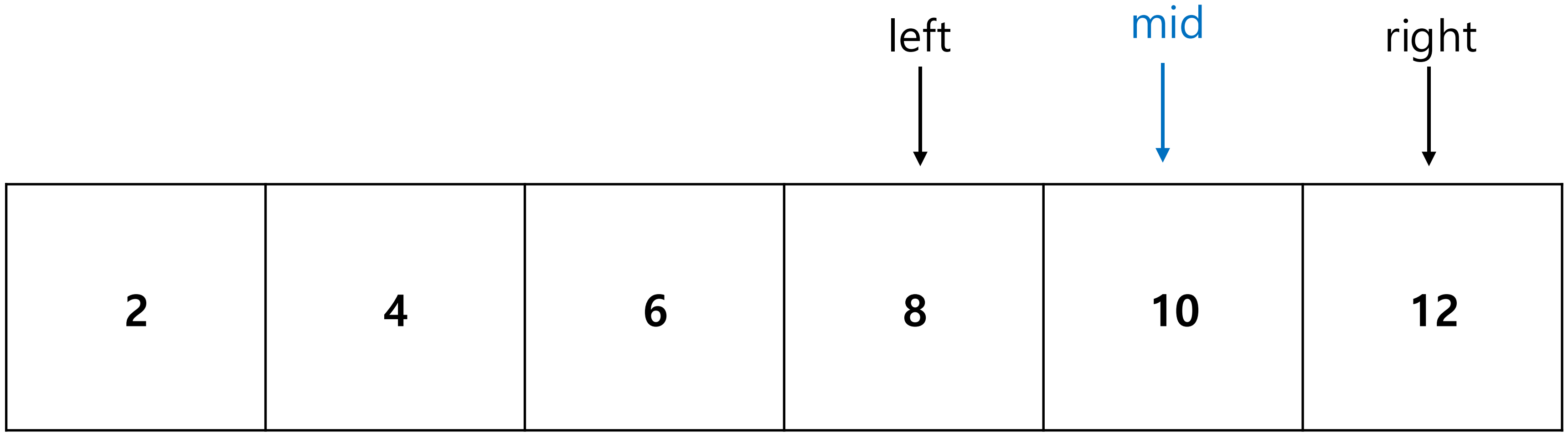
현재 탐색 범위는 [3, 5]가 된다.



이분탐색 | 01 개요

다시 두 범위의 중간 지점을 찾는다. (3 + 5)/2 = 4

mid에 위치한 값이 10이다. 찾던 값과 일치한다. 해당 값의 인덱스를 반환한다.



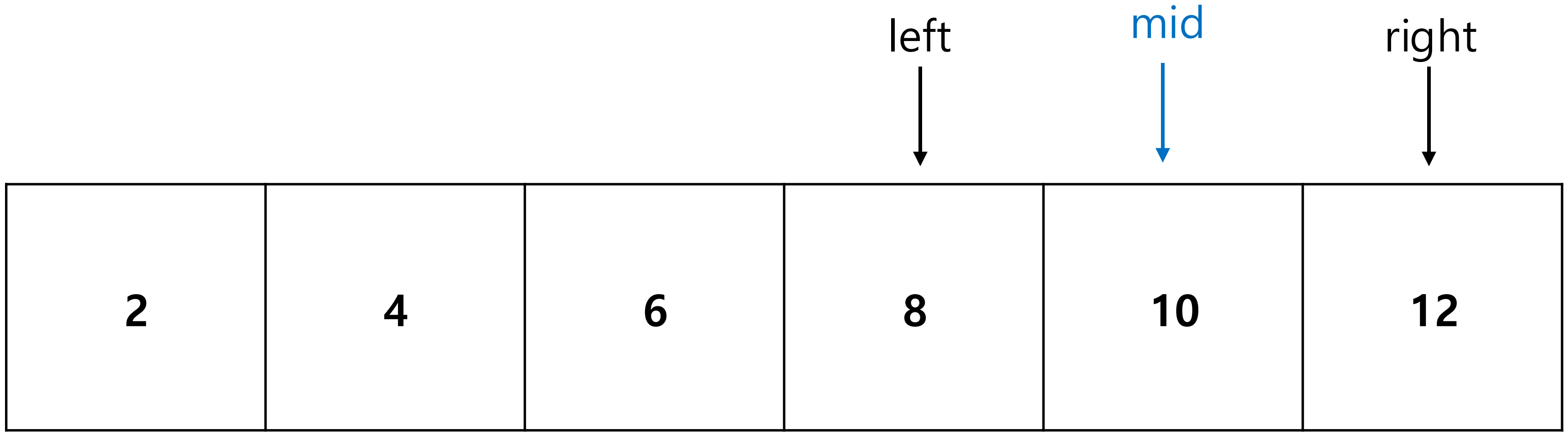
이분탐색 | 01 개요

추가적으로, 만약 배열에 없는 값인 11을 찾는다고 생각해보자.

mid에 위치한 값이 11보다 작다.

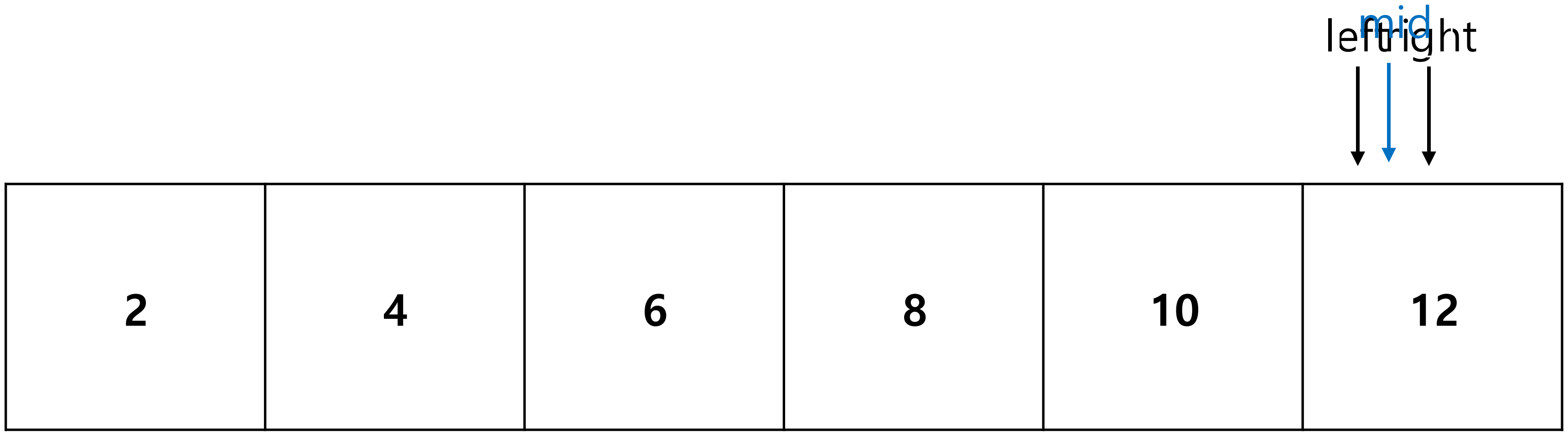
left를 mid + 1 값으로 둔다.

탐색의 범위는 [5, 5]가 된다.



이분탐색 | 01 개요

mid에 위치한 값은 12로, 찾고자 하는 값인 11보다 크다. right를 mid – 1 값으로 둔다.



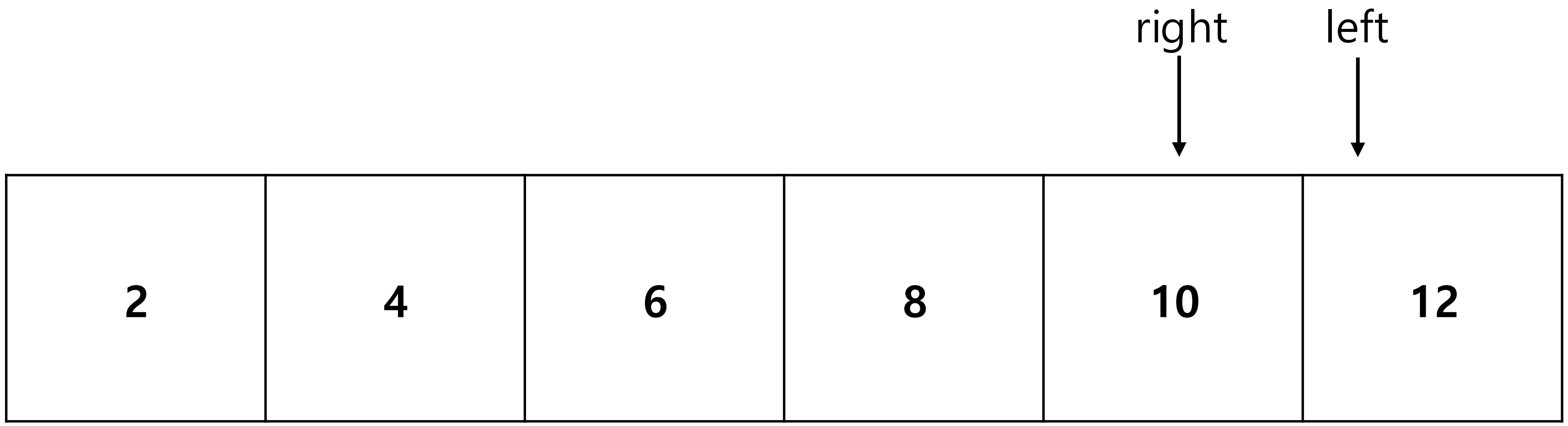
이분탐색 | 01 개요

right를 mid – 1 값으로 두면, 탐색 범위가 유효하지 않다.

right < left 가 되버렸기 때문이다.

찾고자 하는 값을 못 찾았다는 것이다.

오류의 표시로 -1를 반환하면 된다.



이분 탐색 | 02 구현

- 반복문으로 구현한 이분 탐색

```cpp
int binary_search(vector<int> &arr, int target){
    int left = 0, right = arr.size() - 1;

    while(left <= right){
        int mid = (left + right) >> 1;
        if(arr[mid] == target){
            return mid;
        }else if(arr[mid] < target){ // target이 더 크면 오른쪽 탐색
            left = mid + 1;
        }else{ // target이 더 작으면 왼쪽 탐색
            right = mid - 1;
        }
    }
    return -1;
}
```

이분 탐색 | 02 구현

- 재귀로 구현한 이분탐색

(함수 정의)

binary_search(arr, left, right, target) : 배열 arr에서 구간 [left, right]에 대해 target값이 있는지 확인하고 없다면 -1, 있다면 해당 원소의 인덱스를 반환한다.

(기저 조건) left > right인 경우 구간을 벗어나므로 값이 존재하지 않는다는 뜻이다. -1를 반환한다.

구간 중간에 위치한 값이 목표한 값과 같으면 해당 원소의 인덱스를 반환한다.

그렇지 않은 경우에는 arr[mid]와 target의 대소관계에 따라 다음을 수행한다.

arr[mid] > target 인 경우

배열 arr에서 구간 [left, mid – 1]에 대해 target값이 있는지 확인하고 없다면 -1, 있다면 해당 원소의 인덱스를 반환한다 **(재귀!)**

arr[mid] < target 인 경우

배열 arr에서 구간 [mid + 1, right]에 대해 target값이 있는지 확인하고 없다면 -1, 있다면 해당 원소의 인덱스를 반환한다 **(재귀!)**

이분 탐색 | 02 구현

- 재귀로 구현한 이분 탐색

```cpp
int binary_search(vector<int> &arr, int left, int right, int target){
    if(left > right) return -1;

    int mid = (left + right) >> 1;
    if(arr[mid] == target) return mid;
    else if(arr[mid] > target){
        return binary_search(arr, left, mid - 1, target);
    }else{
        return binary_search(arr, mid + 1, right, target);
    }

}
```

이분 탐색 | 03 lower_bound, upper_bound

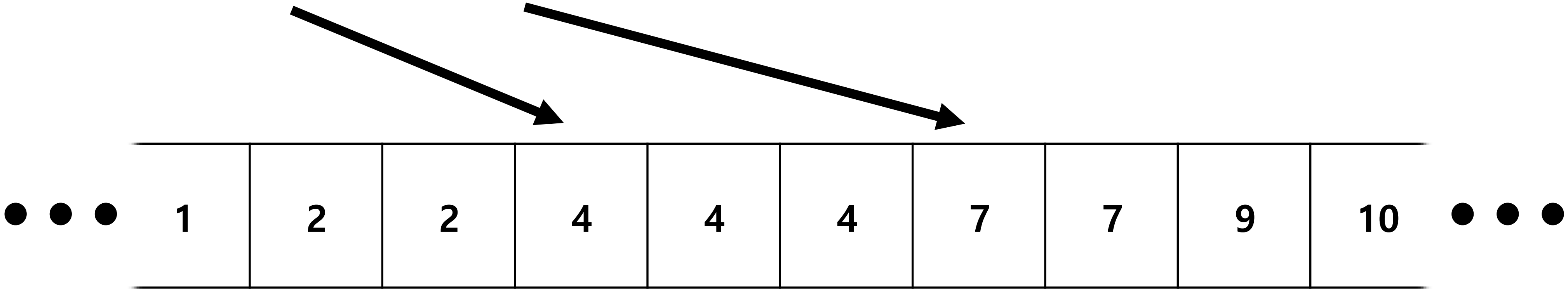
- 개념

이분 탐색을 활용하는 과정에서 특정 값을 찾는 것뿐만 아니라, 특정 조건을 만족하는 첫 번째 또는 마지막 위치를 찾는 경우가 많다. 이때 lower_bound (하한) 과 upper_bound (상한) 이라는 개념이 도입된다.

lower_bound란, 주어진 값 x 이상이 처음 등장하는 위치를 찾는 것이다. (하한)

upper_bound란, 주어진 값 x 초과가 처음 등장하는 위치를 찾는 것이다. (상한)

예를들어 lower_bound(4), upper_bound(4)의 결과 값은 다음과 같다. 당연히 배열은 정렬되어 있어야 한다.



이분 탐색 | 03 lower_bound, upper_bound

- C++ STL

C++ STL에서는 std::lower_bound와 std::upper_bound가 기본적으로 제공된다.

lower_bound(first, last, value)

first, last : 구간 [first, last) 범위에서 탐색할 정렬된 배열의 시작과 끝 반복자

value : 찾고자 하는 값

반환값 : value 이상인 첫 번째 원소의 반복자를 반환한다. 찾지 못하면, last 를 반환한다.

upper_bound(first, last, value)

first, last : 구간 [first, last) 범위에서 탐색할 정렬된 배열의 시작과 끝 반복자

value : 찾고자 하는 값

반환값 : value 초과인 첫 번째 원소의 반복자를 반환한다. 찾지 못하면, last 를 반환한다.

이분 탐색 | 03 lower_bound, upper_bound

- 사용법

위 함수들은 반복자를 반환하므로, arr.begin()를 빼주면 해당 원소의 인덱스를 얻을 수 있다.

```
int lower = lower_bound(arr.begin(), arr.end(), x) - arr.begin();
int upper = upper_bound(arr.begin(), arr.end(), x) - arr.begin();
```

참고) upper – lower 을 계산해서 값 x가 몇 개 있는지 계산할 수 있다.

이분 탐색 | 04 예제 : 10816 숫자 카드 2

- 문제 요약

  정수 카드를 여러 장 가지고 있다. 주어진 정수 카드 목록에서 특정 정수가 몇 개 있는지 찾아야 한다.

- 입력
  - 첫 번째 줄 : 숫자 카드의 개수 N (1 <= N <= 500,000)
  - 두 번째 줄 : N개의 숫자카드 (수의 범위 : -10^7 ~ 10^7)
  - 세 번째 줄 : 찾고 싶은 숫자의 개수 (1 <= M <= 500,000)
  - 네 번째 줄 : 찾고 싶은 숫자 M개 (수의 범위 : -10^7 ~ 10^7)

예제 입력 1 복사

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

- 출력
  - M개의 숫자 각각이 숫자 카드에 몇 개 존재하는지 출력한다.

이분 탐색 | 04 예제 : 10816 숫자 카드 2

- 풀이

가장 단순한 접근방법은 각각의 쿼리에 대해 배열을 훑어보는 방식. O(NM)이다.

N, M의 최댓값이 500,000이므로 짜면 당연히 TLE이다.

예제 입력 1 복사

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

어떻게 할까?

배열을 정렬해볼까?

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

이분 탐색 | 04 예제 : 10816 숫자 카드 2

- 풀이

쿼리가 들어올 때마다 다음을 수행하면 될 것 같은데?

(x 초과인 첫 번째 원소의 위치) – (x 이상인 첫 번째 원소의 위치)

예제 입력 1 복사

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

예를 들어, x = 3의 경우

upper_bound와 lower_bound를 사용하면 되겠다!

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

이분 탐색 | 04 예제 : 1654번 랜선 자르기

- 구현

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, M;
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);

    cin >> N;
    vector<int> v(N);
    for(int i = 0; i < N; i++) cin >> v[i];
    sort(v.begin(), v.end());
    cin >> M;
    int tmp;
    while(M--){
        cin >> tmp;
        cout << upper_bound(v.begin(), v.end(), tmp) - lower_bound(v.begin(), v.end(), tmp) << " ";
    }
    return 0;
}
```

매개변수 탐색

매개변수 탐색 | 01 개요

- 매개변수 탐색은 **결과를 판별할 수 있는 함수**를 정의하고, 이를 기반으로 **특정 범위에서 최적의 값을 찾는 것**이 핵심이다.
- 매개변수 탐색은 보통 **결정 문제**를 기반으로 동작한다.
    - 결정 문제는 Yes / No 로 답할 수 있는 문제를 말한다.

매개변수 탐색 | 01 개요

- 매개변수 탐색의 기본적인 형태

  특정 값이 조건을 만족하는지 확인하는 check() 함수를 만드는 건 필수!

  1. 탐색할 범위 [left, right]를 정의한다.
  2. mid = (left + right) / 2 값을 계산한다.
  3. mid 값이 조건을 만족하는 지, 만족하지 않는지 확인하고 탐색의 범위를 조절한다.

     check(mid) == True; // 조건을 만족하는 값이므로 더 좋은 해를 찾기 위해 탐색 범위를 조정한다.

     check(mid) == False; // 조건을 만족하지 않는 값이므로 탐색 범위를 조정한다.

  4. 최적의 값을 찾을 때까지 위 과정을 반복한다.

매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

- 문제 요약

  길이가 제각각인 K개의 랜선을 잘라서 N개의 같은 길이의 랜선을 만들려고 한다. N개보다 많이 만드는 것도 N개를 만드는 것에 포함된다. 이미 자른 랜선은 다시 붙일 수 없다. 만들 수 있는 랜선의 최대 길이를 구하자.

- 입력

  K (현재 가지고 있는 랜선 개수) N (필요한 랜선 개수)

  {K개의 랜선 길이 값이 차례대로 주어진다.}

- 출력

  랜선의 최대 길이를 출력한다.

1 <= K <= 10,000

1 <= N <= 1,000,000

1 <= K <= 2^31 - 1

예제 입력 1 복사

```
4 11
802
743
457
539
```

예제 출력 1 복사

```
200
```

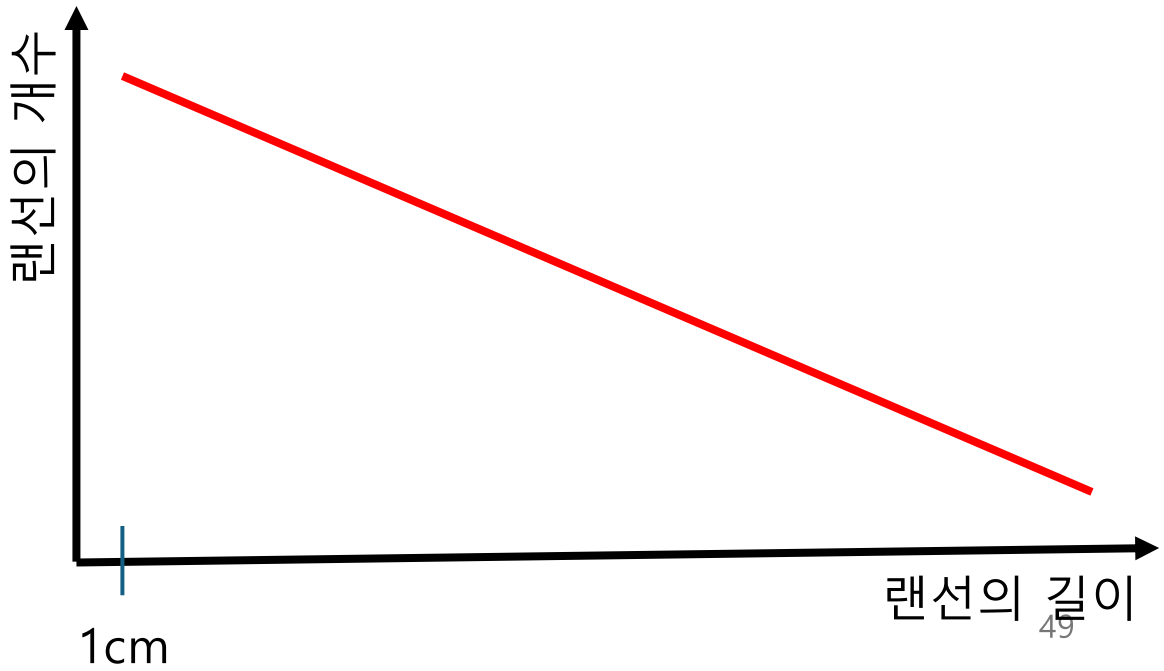
매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

- 풀이

결국은 K개의 랜선을 잘라서 N개의 같은 길이의 랜선을 만드는 것이고, 그 길이를 출력하는게 목표이다.

한 가지 떠오르는 방법은 **가장 작은 길이인 1cm** 부터 **K개의 랜선 중 가장 긴 것의 길이**까지 잘라보면서 **몇 개의 랜선이 나오는지 확인**하는 것이다.

그걸 함수로 나타내면 이런 경향이 나타날 것 같다.



참고) 모든 경우를 탐색할때의 시간복잡도

O(랜선의 최대길이 x 랜선의 개수) ≈ 2 * 10^13 이므로

1초에 1억(1e8)을 훨씬 초과! **시간 초과!!!**

매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

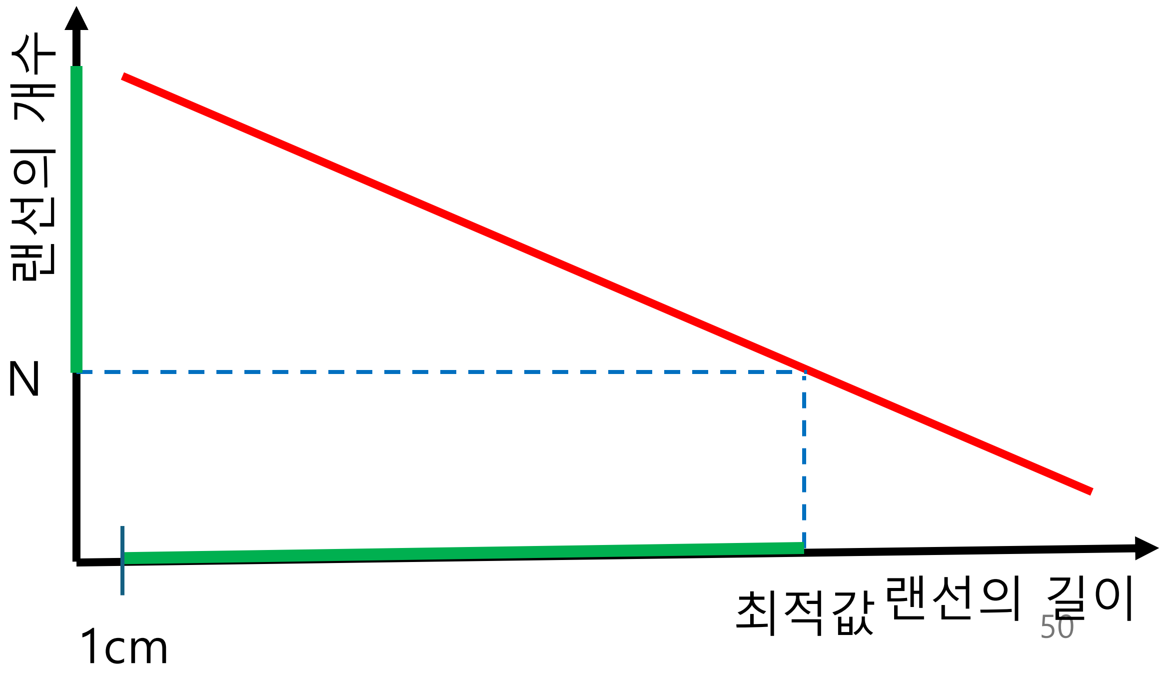
- 풀이

길이가 같은 랜선을 N개 이상 만들어야 하기 때문에
특정한 값을 기준으로 랜선을 N개 이상 만들 수 있는지 없는지 결정된다.

True와 False의 경계에 있는 특정한 값은 최적값을 의미하고, 즉 문제에서 구해야하는 답이 된다는 걸 알 수 있다.

최적값을 기준으로 Yes / No 가 결정된다는 것을 주목하면 좋다.

그림에서 초록색으로 칠한 부분은 Yes다.



매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

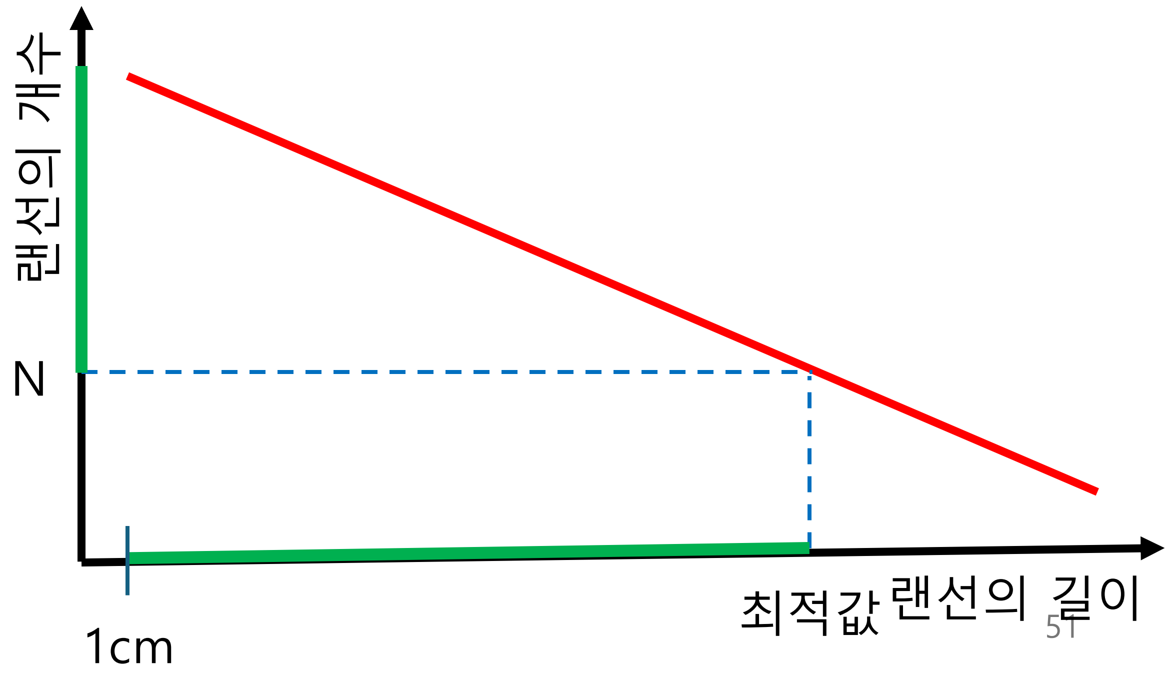
- 풀이

  랜선의 길이를 인자로 넣으면 랜선을 N개 이상 만들 수 있는지/없는지 알려주는 함수 check(length)을 정의하자.

  굳이 랜선을 모든 길이로 자르지 않아도 된다는 것을 주목해야 한다.

  이분탐색의 아이디어를 활용해서 중간값에 대해 가능, 불가능 여부에 따라 범위를 조정하면 된다.

  이게 매개변수 탐색의 흐름이다.



매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

- 구현

Line 8 ~ 15

랜선의 길이를 인자로 넣으면 랜선을 N개 이상 만들 수 있는지

알려주는 check 함수이다.

주어진 랜선을 길이 l 로 잘라보면서 N개 이상이면

true를 반환하고 그렇지 않으면 false를 반환한다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  #define ll long long
4
5  ll N, K, ans;
6  vector<ll> LAN;
7
8  bool check(int l){
9      int lan = 0;
10     for(ll &x : LAN){
11         lan += x / l;
12         if(lan >= N) return true;
13     }
14     return false;
15 }
```

매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

- 구현

Line 23

탐색할 범위를 설정한다.

Line 24 ~ 32

매개변수 탐색을 진행한다.

check(mid)가 참이면, mid 보다 길게 잘라도

된다는 의미니 left = mid + 1를 해준다.

거짓이면 mid 보다 짧게 잘라야 하니

right = mid – 1를 해준다.

```
17  int main(){
18      ios::sync_with_stdio(false);
19      cin >> K >> N;
20      LAN.resize(N);
21      for(int i = 0; i < K; i++) cin >> LAN[i];
22  
23      ll left = 1, right = *max_element(LAN.begin(), LAN.end());
24      while(left <= right){
25          ll mid = (left + right) >> 1;
26          if(check(mid)){
27              ans = mid;
28              left = mid + 1; // mid cm보다 더 길게 잘라도 됨.
29          }else{
30              right = mid - 1; // mid cm보다 짧게 잘라야 됨.
31          }
32      }
33      cout << ans << "\n";
34      return 0;
35  }
```

분할 정복

분할 정복 | 01 개요

- 분할 정복(Divide and Conquer) 알고리즘은 문제를 작은 부분으로 나누어 해결한 후, 이를 합쳐 전체 문제의 해답을 얻는 알고리즘 패러다임이다.
- 이 방식은 큰 문제를 작은 문제로 나누는 재귀적 접근법을 기반으로 동작한다.

  예) 병합 정렬, 퀵 정렬, 이진 탐색, 거듭 제곱, 최근접 점 쌍 문제

분할 정복 | 01 개요

- 분할 정복의 기본 구조

1. 분할(Divide): 문제를 더 작은 하위 문제로 나눈다.
2. 정복(Conquer): 하위 문제를 해결한다. 보통 이 단계에서 재귀 호출 or 반복문이 사용된다.
3. 합병(Combine): 하위 문제들의 결과를 결합하여 원래 문제의 해를 구한다.

분할 정복 | 02 예제 (1) : 병합 정렬

- 특징

  분할 정복 알고리즘을 기반으로 동작한다.

  배열을 반으로 나누고, 각각을 정렬한 후 합치는 과정을 반복하여 정렬을 수행한다.

  합치는 과정에서 추가적인 메모리가 필요하다.

- 동작 과정

  1. 배열을 절반으로 나눈다. 더 이상 나눌 수 없을 때 까지 나눈다. **(재귀!)**
  2. 각각의 작은 배열들을 정렬하며 병합한다.

분할 정복 | 02 예제 (1) : 병합 정렬

- 병합 정렬에서 나타나는 분할 정복

1. 분할 (Divide)

   입력 배열을 같은 크기의 부분 배열 2개로 분할한다.

   이 과정을 배열의 크기가 1이 될 때까지 재귀적으로 반복한다.

2. 정복 (Conquer)

   배열의 크기가 1이면 이미 정렬된 상태이다.

   분할된 각각의 부분 배열을 정렬한다.

3. 병합 (Merge)

   투 포인터 알고리즘을 활용해 정렬된 두 개의 부분 배열을 하나의 정렬된 배열로 합친다.

* 구현은 이전에 다뤘으므로 넘어가겠습니다.

분할 정복 | 02 예제 (1) : 병합 정렬

- 구현

Line 5. merge_sort(arr, left, right) 라는 함수는 구간 [left, right] 를 정렬해주는 알고리즘이라고 **믿기(정의하자).**

Line 6. left == right인 순간은 원소가 하나만 있는 상태이다. **그 자체로 정렬된 상태**인 것이다. 기저 조건인 것이다.

Line 9 ~ 11. 그 구간의 절반을 나눈다. 이후에 merge_sort(arr, left, mid)와 merge_sort(arr, mid + 1, right)를 호출한다. (**재귀!**)

Line 12. **11행으로 실행이 넘어오면 구간 [left, mid], [mid + 1, right]는 각각 정렬되어 있으므로 잘 합쳐 주기만 하면 된다.**

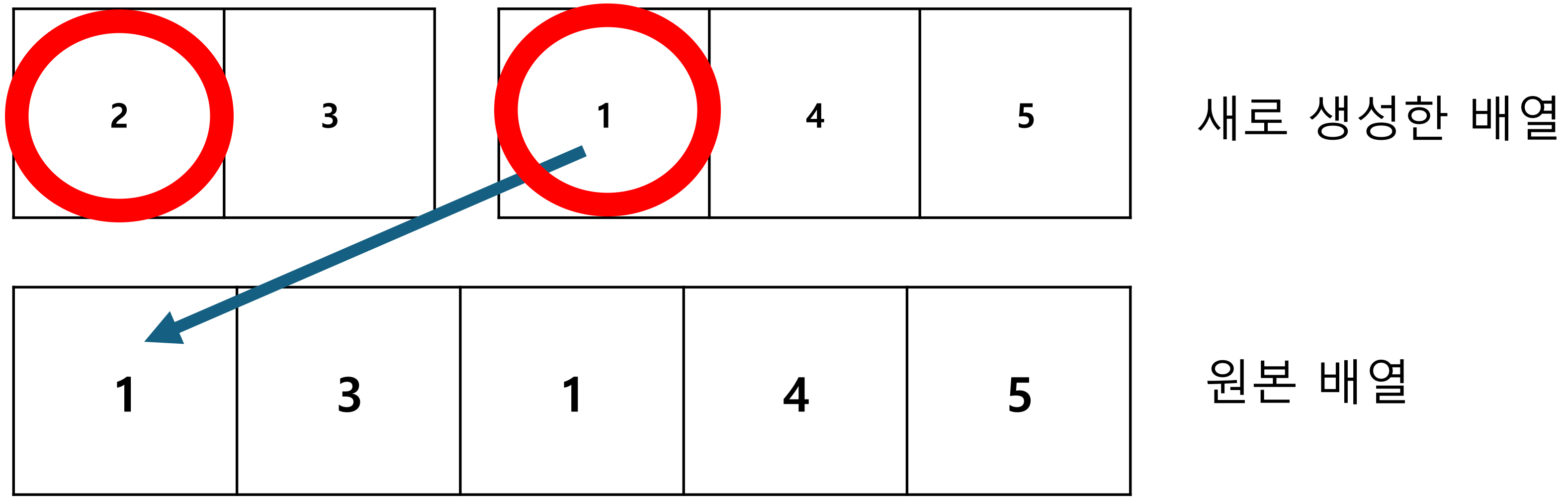
```
5   void merge_sort(vector <int> &arr, int left, int right){
6       if(left >= right) return; // 원소가 하나만 남으면 종료
7
8       int mid = (left + right) >> 1;
9       merge_sort(arr, left, mid);
10      merge_sort(arr, mid + 1, right);
11      merge(arr, left, mid, right);
12      return;
13  }
```

분할 정복 | 02 예제 (1) : 병합 정렬

- 구현

정렬된 두 부분 배열 [left, mid], [mid + 1, right]를 합쳐주는 방법? 투 포인터 기법을 활용하면 된다!

1. 새로운 배열 생성하고 원본 배열 값 복사하기.
2. 두 배열의 첫 번째 원소부터 비교하여 더 작은 값을 원본 배열을 수정하기
3. 하나의 배열이 먼저 끝나면, 남아 있는 원소들을 그대로 새로운 배열에 추가하기



분할 정복 | 02 예제 (1) : 병합 정렬

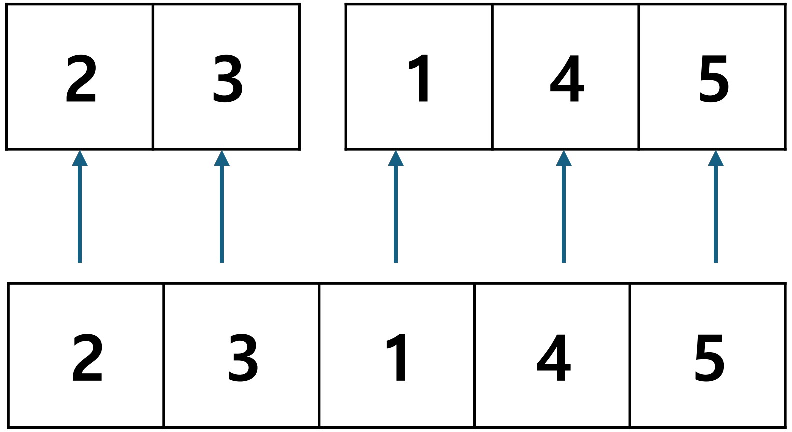
- 구현

1. 새로운 배열 생성하고 값을 복사하기

```cpp
void merge(vector<int> &arr, int left, int mid, int right){
    int left_size = mid - left + 1; // [left, mid]
    int right_size = right - mid; // [mid + 1, right]

    vector<int> left_arr(left_size);
    vector<int> right_arr(right_size);

    // left_rr, right_arr에 데이터 복사
    for(int i = 0; i < left_size; i++){
        left_arr[i] = arr[left + i];
    }
    for(int j = 0; j < right_size; j++){
        right_arr[j] = arr[right + j];
    }
```



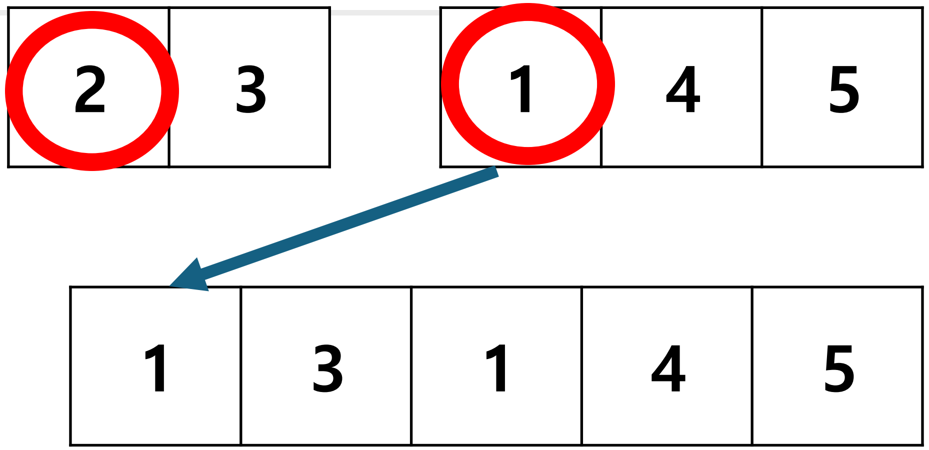
분할 정복 | 02 예제 (1) : 병합 정렬

- 구현

2. 두 배열의 첫 번째 원소부터 비교하여 더 작은 값을 새로운 배열에 추가하기

```
    // i : left_arr 인덱스, j : right_arr 인덱스, k : arr 인덱스
    int i = 0, j = 0, k = left;
    while(i < left_size && j < right_size){
        if(left_arr[i] <= right_arr[j]){
            arr[k++] = left_arr[i++];
        }else{
            arr[k++] = right_arr[j++];
        }
    }

```

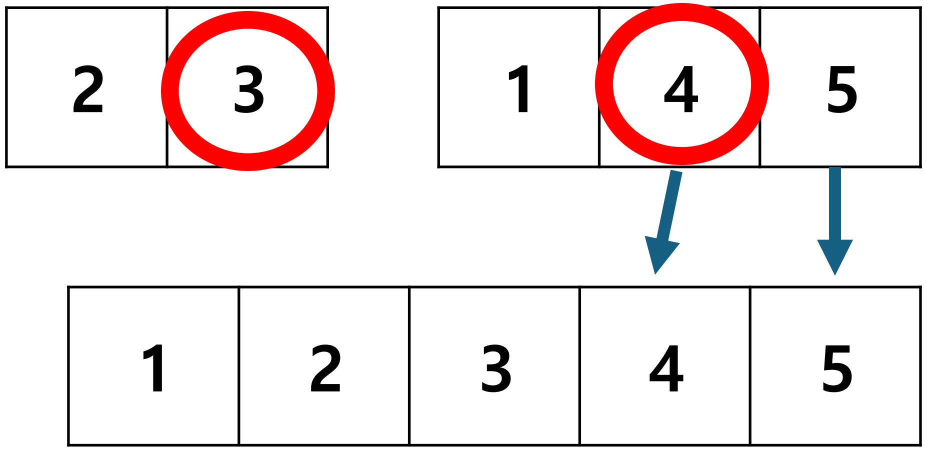


분할 정복 | 02 예제 (1) : 병합 정렬

- 구현

3. 하나의 배열이 먼저 끝나면, 남아 있는 원소들을 그대로 새로운 배열에 추가하기

```
    // 남아 있는 요소들 복사
    while(i < left_size){
        arr[k++] = left_arr[i++];
    }
    while(j < right_size){
        arr[k++] = right_arr[j++];
    }
}
```



분할 정복 | 02 예제 (2) : 백준 1629번 빠른 거듭제곱

- 문제요약

  A^B를 구하는 게 목표인데, 매우 커질 수 있으므로 C로 나눈 나머지를 출력해라.

  A, B, C는 2,147,483,647 이하의 자연수이다.

- 입력

  A B C가 주어진다.

- 출력

  첫째 줄에 A를 B번 곱한 수를 C로 나눈 나머지를 출력한다.

예제 입력 1 복사

```
10 11 12
```

예제 출력 1 복사

```
4
```

분할 정복 | 02 예제 (2) : 백준 1629번 빠른 거듭제곱

- 풀이
  - A^B mod C를 직접 계산해볼까?
  - A^B 를 계산하고 C로 나눈 나머지를 구하면 되는데…
  - 이 과정에서 오버플로우가 발생하여 틀린 값이 나온다.
    - 오버플로우가 나지 않더라도 곱하는 연산을 20억번 반복하는 것 자체가 시간 초과의 원인이 될 수 있다.

분할 정복 | 02 예제 (2) : 백준 1629번 빠른 거듭제곱

- 풀이
  - 이 문제를 풀기 위해서는 모듈러 연산의 특징을 활용해야 한다.

$$(A \times B) \bmod C = ((A \bmod C) \times (B \bmod C)) \bmod C$$

"A와 B를 곱한 후 C로 나눈 나머지는, 각각 A와 B를 C로 나눈 나머지를 곱한 후 다시 C로 나눈 나머지와 같다."

이 연산을 활용하면 큰 수의 연산에서 오버플로우를 방지하는 데 유용하다. 즉, 값이 훼손되지 않는다.

분할 정복 | 02 예제 (2) : 백준 1629번 빠른 거듭제곱

- 풀이 $$(A \times B) \bmod C = ((A \bmod C) \times (B \bmod C)) \bmod C$$
  - A^2 mod C의 결과로 (A^2 * A)mod C를 하고, 그것의 결과로 (A^3 * A) mod C를 하면

    오버플로우가 나지 않을 것이므로 올바른 A^B mod C 의 결과값이 나온다.

    하지만, B는 여전히 21억이므로 저 연산을 21억번 반복하는건 시간초과의 여지가 있어 보인다.

분할 정복 | 02 예제 (2) : 백준 1629번 빠른 거듭제곱

- 풀이 $$(A \times B) \bmod C = ((A \bmod C) \times (B \bmod C)) \bmod C$$
  - 분할 정복의 아이디어를 활용하면 B번을 전부 하지 않아도 된다.
  - 예를 들자면, $A^{17} mod\ C = (A\ mod\ C) * (A^8\ mod\ C)^2\ mod\ C$ 로 계산할 수 있다.
  - $A^{17} mod\ C$ 라는 문제를 하위 문제 $A^8 mod\ C$로 분할했다.
  - 이 과정을 반복하다보면 $A^0 mod\ C$ 를 계산하게 되는데 1 mod C는 1이다. (기저 조건)
  - 정복 과정은 간단하다. 구한 $A^8 mod\ C$ 값을 이용해 $A^{17} mod\ C$ 를 계산하면 된다.

Case 1. B가 짝수일 때 $A^B\ mod\ C = \left(A^{\frac{B}{2}}\right)^2 mod\ C$

Case 2. B가 홀수일때 $A^B\ mod\ C = A\ * \left(A^{\frac{(B-1)}{2}}\right)^2 mod\ C$

분할 정복 | 02 예제 (2) : 백준 1629번 빠른 거듭제곱

- 구현 (재귀)

Line 7 ~ 8

- func(b)는 a^b mod c를 계산해주는 함수이다.
- (기저조건) b = 0인 경우, a^0 mod C = 1이다.

Line 10

- 편의를 위해 $\left(a^{\frac{b}{2}}\right)^2\ mod\ C$ 를 먼저 계산해준다.

  b가 홀수이든, 짝수이든 항상 쓰이는 값이기 때문이다.

Line 12 ~ 17

- 짝수일 때, 홀수일 때 값을 계산해준다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  #define ll long long
4  
5  ll A, B, C;
6  
7  ll func(ll b){
8      if(b == 0) return 1; // a^0은 1이다.
9      ll half = func(b/2); // a^(b/2)을 계산한다.
10     ll all = half * half % C; // (a^(b/2))^2을 계산한다.
11 
12     if(b % 2 == 1){
13         // b가 홀수이면 a^n = a * a^(b-1) = a * (half^2)이다.
14         return (A * all) % C;
15     }
16     // b가 짝수이면 a^b = (a^(b/2))^2 = half^2 이다.
17     return all;
18 }
19 
20 int main(){
21     cin >> A >> B >> C;
22     cout << func(B) << "\n";
23     return 0;
24 }
```

문제

- **정렬, 투 포인터, 이분 탐색, 매개변수 탐색, 분할 정복 문제를 푸시면 됩니다!**
- **기본**
  - 백준 1181번 단어 정렬
  - 백준 20922번 겹치는 건 싫어
  - 백준 2630번 색종이 만들기
  - 백준 2776번 암기왕
  - 백준 24343번 기타 레슨
- **심화**
  - 백준 1074번 Z
  - 백준 1992번 쿼드트리
  - 백준 2473번 세 용액

추천하는 문제집

https://www.acmicpc.net/workbook/view/7317
https://www.acmicpc.net/workbook/view/7318
https://www.acmicpc.net/workbook/view/8400
https://www.acmicpc.net/workbook/view/8709

오늘 스터디는 여기서 마무리하겠습니다.

백준 많이 푸세요! 수고하셨습니다!