

OT

25.03.24 (월) 오후 5시 ~ 7시

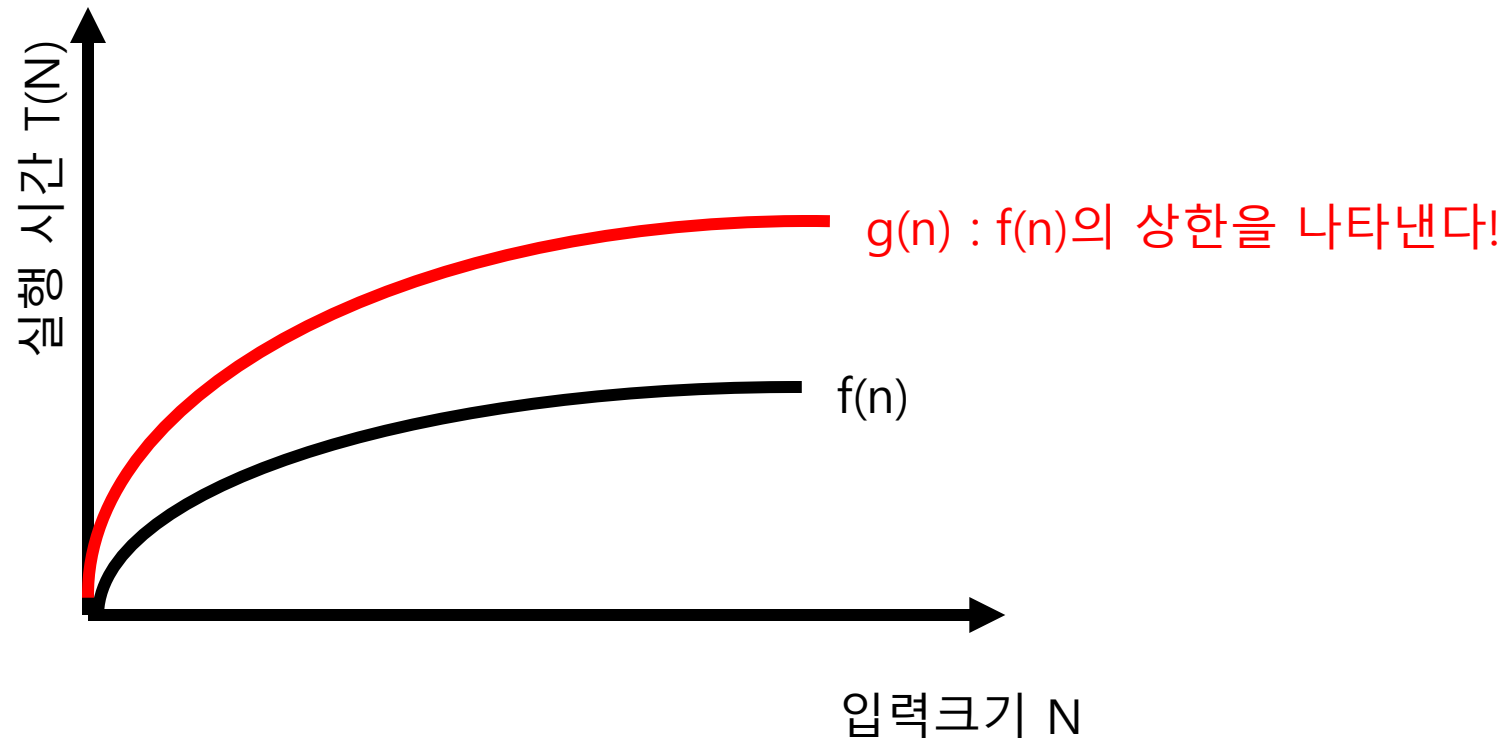
목차

- 시간 복잡도
- 정렬
- 투 포인터
- 이분 탐색
- 매개변수 탐색 (시간이 남으면 진행 or 3차시에 다룸)
- 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 경우(상한)를 나타내 주기 때문이다.



시간 복잡도 | 02 빅오(Big-O) 표기법

- 수학적 정의

어떤 함수 $f(n)$ 에 대해, 다음 조건을 만족하는 양의 상수 c 와 n_0 가 존재하면 $f(n)$ 은 $O(g(n))$ 이라고 한다.

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \geq 0 \text{ such that } |f(n)| \leq c * |g(n)|, \forall n \geq n_0\}$$

$O(g(n))$ 는 특정 함수 $g(n)$ 의 상수 배보다 빠르게 증가하지 않는 모든 함수들의 집합이라는 의미이다.

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.

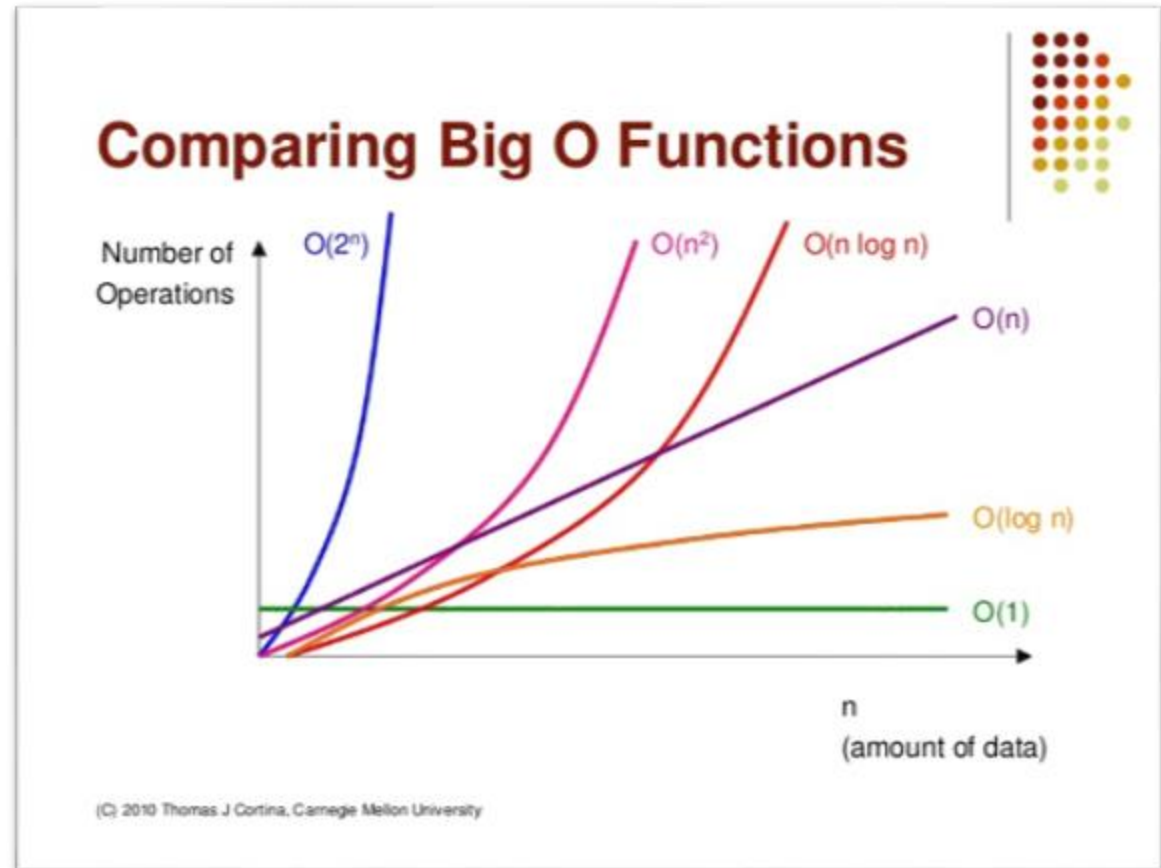
ex) $f(n) = 2^n + 10n$ 의 시간복잡도는 $O(2^n)$ 이다.

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

시간 복잡도 | 02 빅오(Big-O) 표기법

주어진 시간 복잡도를 빅오 표기법으로 나타내는 연습을 해보자.

- $a(n) = 5n \log n + 10n \rightarrow O(n \log n)$
- $b(n) = 3n^n + n! \rightarrow O(n^n)$
- $c(n) = \log n + 4n \rightarrow O(n)$
- $d(n) = 2n^2 + 10n \rightarrow O(n^2)$
- $e(n) = 2^n + 10n \rightarrow O(2^n)$
- $f(n) = 2n^3 + 10n \rightarrow O(n^3)$
- $g(n) = 10 \log n + 4 \rightarrow O(\log n)$



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($1e8$) ~ 10억번($1e9$) 동작한다고 생각하면,

- "N이 5,000이면 $O(N^2)$ 까지는 가능하겠구나." // $N^2 = 2.5 * 10^7$
- "N이 200,000이면 $O(N \log N)$ 정도의 알고리즘을 짜야겠구나." // $200,000 * 17 = 3.4 * 10^6$

라는 생각을 할 수 있다.

시간 복잡도 | 03 예시

- 배열의 원소 접근
 - 배열에서 k 번째 원소의 메모리 주소는 다음과 같이 계산한다.
(배열의 시작 주소) + $k * (\text{자료형의 크기})$
 - 두 번의 산술 연산만 필요하므로 시간 복잡도는 $O(1)$ 이다.
- 이중 for문
 - 바깥 루프는 N 번 실행되고, 바깥 루프가 한 번 실행될 때마다 안쪽 루프는 다시 N 번 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 $N - 1$ 회
 - 시간 복잡도 : $T(N) = N - 1$
 - 빅오 표기법으로 나타내면 $O(N)$ 이다.
- 주어진 배열에서 특정한 값 x 가 있는지 탐색하기
 - 연산 : 두 원소의 값 비교
 - 수행 횟수 : 최대 N 회
 - 시간 복잡도 : $H(N) = N$
 - 빅오 표기법으로 나타내면 $O(N)$ 이다.

시간 복잡도 | 04 예제 : 백준 11726번 2 x n 타일링

- 문제요약

- 2 x n 크기의 직사각형을 1 x 2, 2 x 1 타일로 채우는 방법의 수를 구하는 프로그램을 작성하시오.

- 입력

- 첫째 줄에 n이 주어진다. ($1 \leq n \leq 1,000$)

- 출력

- 첫째 줄에 2 x n 크기의 직사각형을 채우는 방법의 수를 10,007로 나눈 나머지를 출력한다.

시간 복잡도 | 04 예제 : 백준 11726번 2 x n 타일링

- 어떤 알고리즘을 작성해야 할까?
 - 2 x n 크기의 직사각형을 1 x 2, 2 x 1 타일로 하나 하나씩 채워봐서 2 x n 크기의 직사각형이 되는지 확인해볼까?
 $O(2^n)$ // n이 최대 1,000이므로 $2^{(1000)}$ 은 대략 1.07×10^{301} -> 당연히 시간초과
 - 다른 방법을 구해야 함.
n = 1,000이라 $O(N^3)$ 을 하회하는 알고리즘 아무거나 작성하면 통과됨.
- 위 문제를 푸는 방법은 DP에서 알아봅시다!

시간 복잡도 | 05 (번외) 공간 복잡도?

- 공간 복잡도는 알고리즘이 실행될 때 사용하는 메모리의 양을 입력 크기(N)에 따라 분석하는 개념
 - 입력 크기(N)와 무관하게 일정한 공간을 사용하는 경우 $\rightarrow O(1)$
 - 입력 크기(N)에 비례하는 공간을 사용하는 경우 $\rightarrow O(N)$
 - 입력 크기(N)의 제곱의 공간을 사용하는 경우 $\rightarrow O(N^2)$

라는 정도만 알고 있으면 ps할 때 문제되지 않는 것 같습니다. **메모리 제한은 여유롭게 주기 때문에 크게 신경 쓸 필요는 없습니다.** 아래 표는 참고만 부탁드립니다.

| 메모리 제한 | 1차원 배열의 최대 크기 | 2차원 배열의 최대 크기 |
|--------|--------------------|------------------|
| 32MB | int arr[8 * 1e6] | int arr[2800^2] |
| 64MB | int arr[16 * 1e6] | int arr[4000^2] |
| 128MB | int arr[32 * 1e6] | int arr[5600^2] |
| 256MB | int arr[64 * 1e6] | int arr[7900^2] |
| 512MB | int arr[128 * 1e6] | int arr[11300^2] |
| 1GB | int arr[256 * 1e6] | int arr[16000^2] |

정렬

정렬 | 01 개요

- 정렬은 주어진 데이터를 특정한 순서로 재배열하는 것을 의미한다.

오름차순 정렬 : $a < b$ 이면 a 가 b 보다 앞에 오도록 나열

내림차순 정렬 : $a > b$ 이면 b 가 a 보다 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.

```
4 void bubble_sort(vector<int> &arr){  
5     for(int i = 0; i < arr.size() - 1; i++){  
6         for(int j = 0; j < arr.size() - i - 1; j++){  
7             if(arr[j] > arr[j + 1]){  
8                 swap(arr[j], arr[j + 1]);  
9             }  
10        }  
11    }  
12 }
```

- 시간 복잡도 분석

벡터 arr의 크기를 n이라 하자.

바깥 루프는 n - 1번 반복한다.

안쪽 루프는 n - 1, n - 2, ..., 1번 반복하며 총 비교 횟수는 $(n - 1) + (n - 2) + \dots + 1 = n * (n - 1) / 2$ 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 함수(comp)가 만족해야 하는 성질**이다.

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

➤ 비반사성(Irreflexivity)

어떠한 원소도 자기 자신보다 작을 수 없다. $x < x$ 는 항상 거짓이어야 한다.

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

➤ 비대칭성(Asymmetry)

어떤 두 원소 a, b 에 대해, $\text{comp}(a, b)$ 가 참이면, $\text{comp}(b, a)$ 는 반드시 거짓이어야 한다.

지켜지지 않으면? 순환 관계가 발생해서 논리적으로 정렬이 불가능해진다. ex) 위상정렬

➤ 추이성(Transitivity)

$\text{comp}(a, b)$ 가 참이고 $\text{comp}(b, c)$ 가 참이면, $\text{comp}(a, c)$ 도 반드시 참이어야 한다.

➤ 동등성의 추이성(Transitivity of Equivalence)

$\text{comp}(a, b)$ 가 거짓이고 $\text{comp}(b, a)$ 도 거짓이면, a 와 b 는 동등한 것으로 간주된다.

이 동등한 관계는 추이적이어야 하며, 다른 원소 c 에 대해서도 $\text{comp}(a, c)$ 와 $\text{comp}(b, c)$ 의 결과는 동일해야 한다.

정렬 | 04 예제 : 백준 11650번 좌표 정렬하기

- 문제 요약

2차원 평면 위의 점 N 개 가 주어진다.

x 좌표를 기준으로 오름차순 정렬해서 출력하기.

x 좌표가 같다면 y 좌표를 기준으로 오름차순 정렬해서 출력하기.

- 시간복잡도

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

- 구현

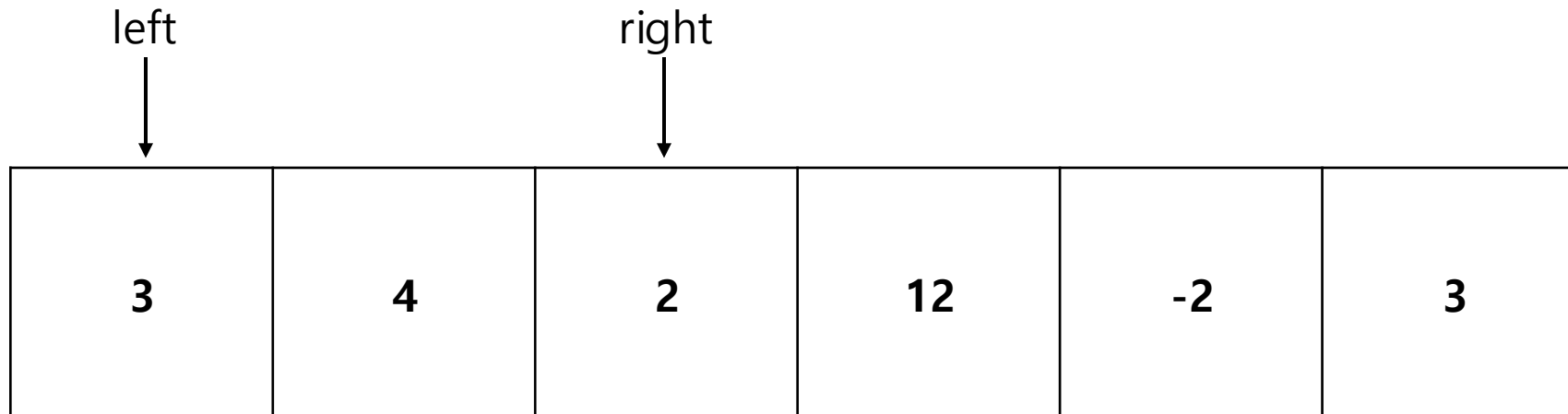
`sort` 함수는 기본적으로 오름차순 정렬을 하기 때문에 따로 함수를 작성해주지 않아도 된다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int N;
5  int main(){
6      cin >> N;
7      vector<pair<int, int>> p(N);
8      for(int i = 0; i < N; i++){
9          cin >> p[i].first >> p[i].second;
10     }
11     sort(p.begin(), p.end());
12     for(auto k : p){
13         cout << k.first << " " << k.second << "\n";
14     }
15     return 0;
16 }
```

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 N 인 수열 A 가 주어진다. ($10 \leq N < 100,000$)

부분 연속된 구간의 합이 s 이상이 되는 가장 짧은 부분 수열의 길이를 출력한다. ($0 < s \leq 100,000,000$)

그러한 수열이 존재하지 않는다면 0을 출력한다.

- 시간 제한

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

N 의 최댓값이 10^5 이므로 $O(N \log N)$ 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.

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

$S[i] = a[0] + a[1] + \dots + a[i - 1]$ // $S[i]$ 는 $a[0]$ 부터 $a[i - 1]$ 까지의 합

이를 이용하면 배열 a 의 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" 조절'해가면 문제의 답을 찾을 수 있겠다'라는 생각이 들 것이다.

투 포인터 기법을 활용할 수 있다.

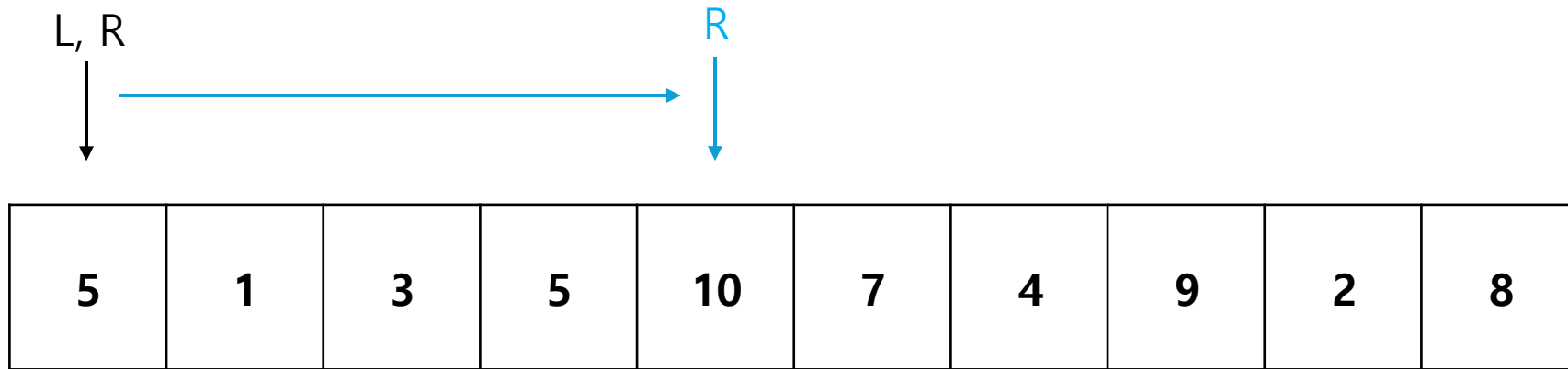
투 포인터 | 02 예제 : 백준 1806번 부분합

- 풀이

L, R을 조절하는 방법?

처음에는 두 개의 포인터 모두 $a[0]$ 에서 시작해본다. 처음에는 L, R은 모두 0 값을 가지고 있고 부분합은 $a[0]$ 이다.

부분합이 15가 넘을 때까지 R을 오른쪽으로 옮겨준다.



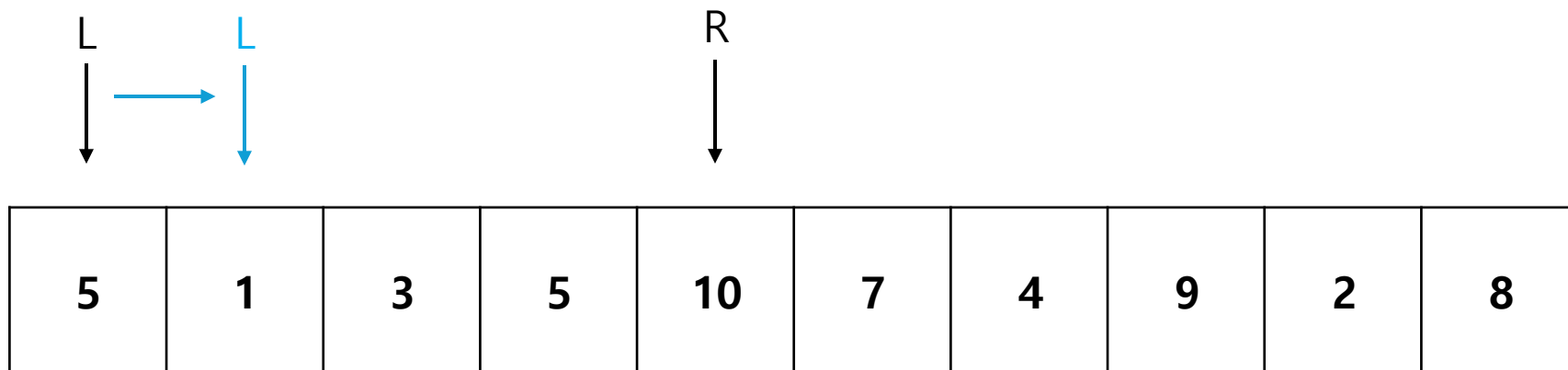
투 포인터 | 02 예제 : 백준 1806번 부분합

- 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.



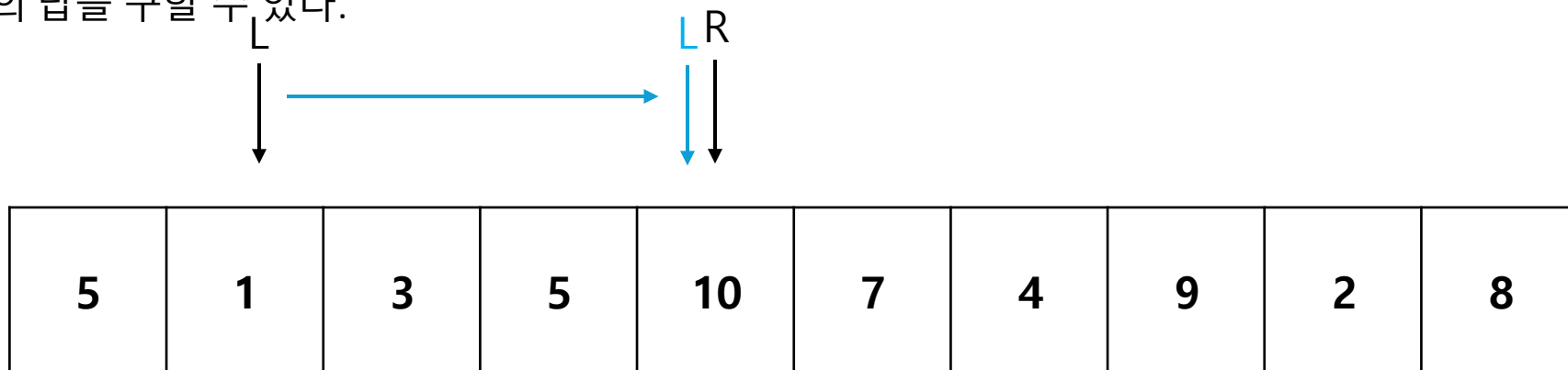
투 포인터 | 02 예제 : 백준 1806번 부분합

- 풀이

s 미만이 될 때까지 L 를 오른쪽으로 옮겨준다. L 을 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 R 이 s 이상일 때까지 움직인다. 위 과정을 L 과 R 이 배열의 끝까지 이동할 때 문제의 답을 구할 수 있다.



투 포인터 | 02 예제 : 백준 1806번 부분합

• 구현

Line 15

L, R은 0으로 초기화되어 있고 sub_sum은 부분합을

저장하는 배열인데 A[0]로 초기화되어 있다.

구간 [0, 0]의 부분합은 A[0]기 때문이다.

Line 16

부분합이 s 미만이어야 하고 오른쪽을 가리키는 포인터

가 배열의 범위를 초과하지 않을 동안 while 문을 반복한다.

Line 19

R이 범위를 벗어나면 더 이상 가능한 수열이 없다는 의미이므로 종료한다.

Line 21

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 brute-force 방식으로 풀게 되면 $O(N^2)$ 인데, 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 (참고만 해주세요)

- 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 (뒤에서 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 것**으로 $O(\log_2 N)$ 의 시간복잡도를 가진다.

이분탐색 | 01 개요

- Up & Down 게임으로 이분 탐색이 어떤 느낌인지 알아봅시다.

 강사:

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

 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:

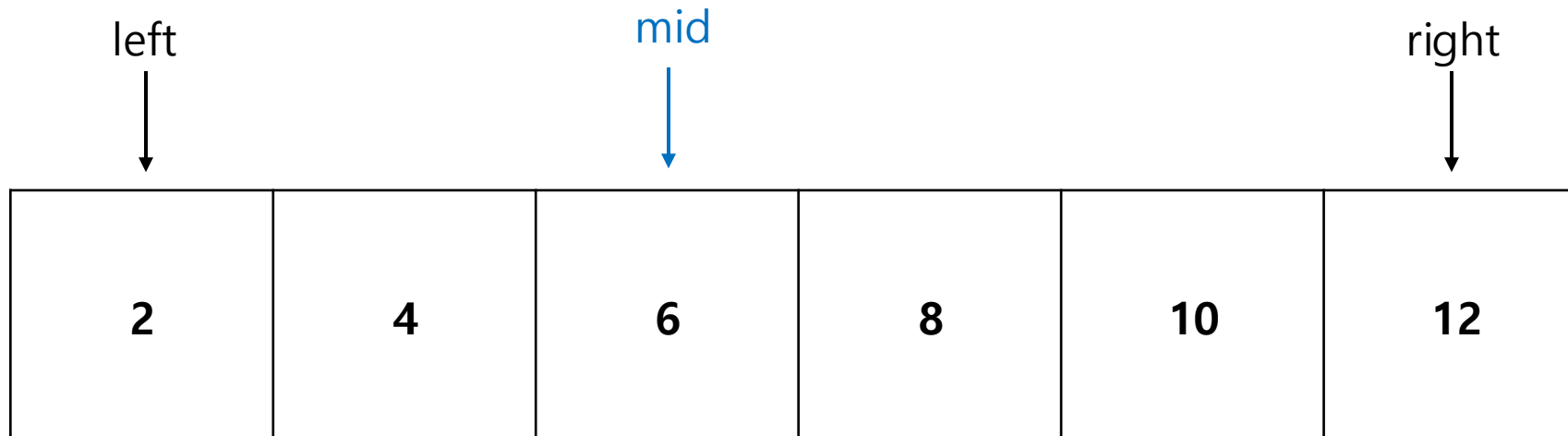
- “맞아요! 정답은 37이었습니다!” 

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 $(0 + 5) / 2 = 2$ 이다.

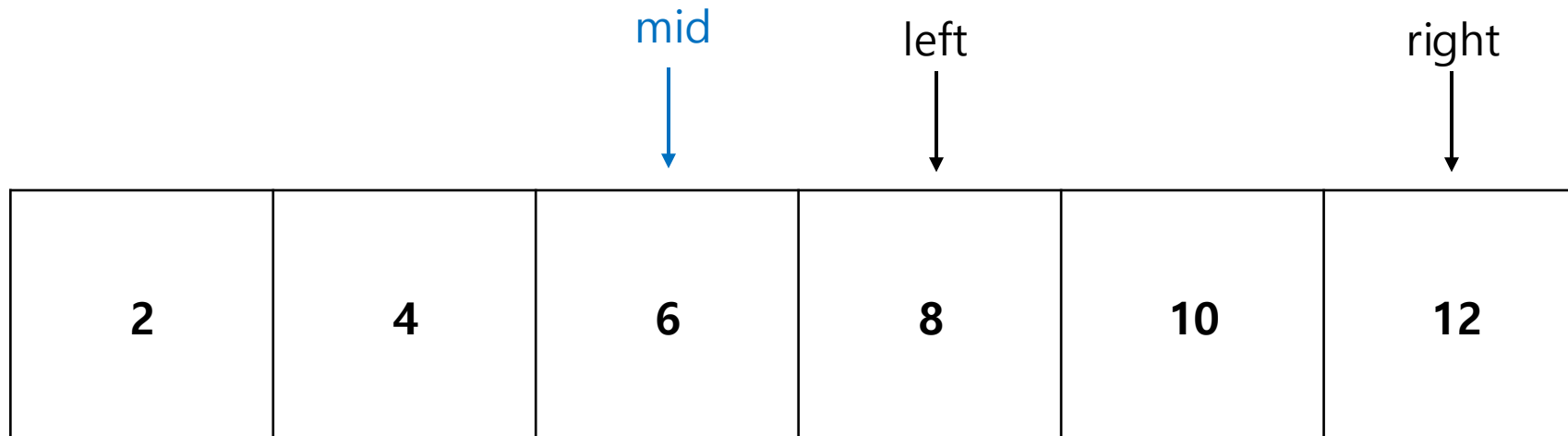


이분탐색 | 01 개요

mid에 위치한 값이 찾고자 하는 값인 10보다 작다.

left를 $\text{mid} + 1$ 로 바꾼다.

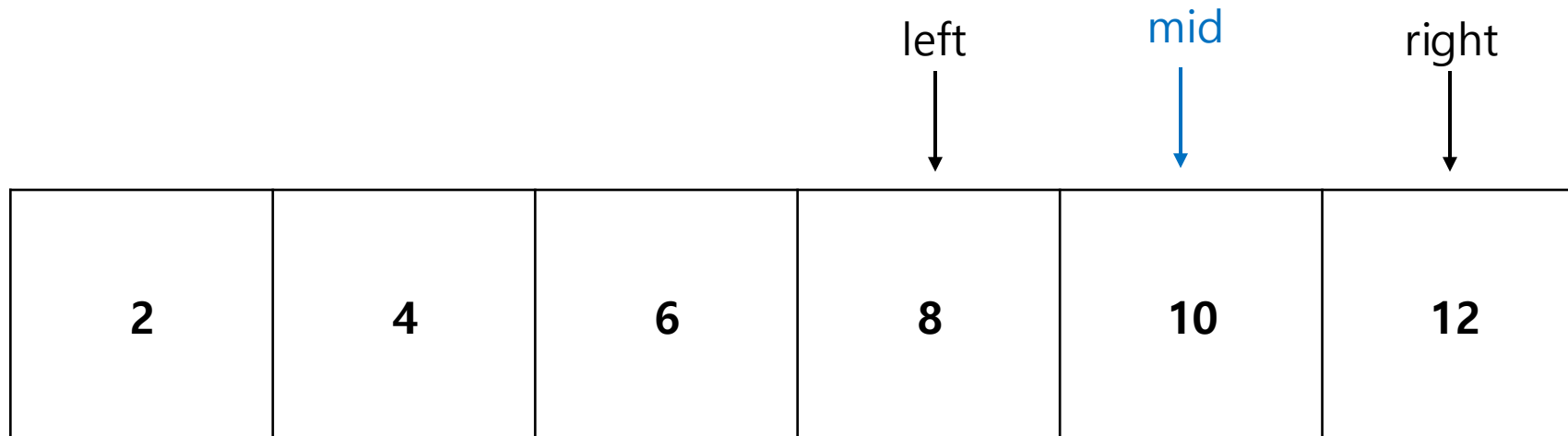
현재 탐색 범위는 $[3, 5]$ 가 된다.



이분탐색 | 01 개요

다시 두 범위의 중간 지점을 찾는다. $(3 + 5)/2 = 4$

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



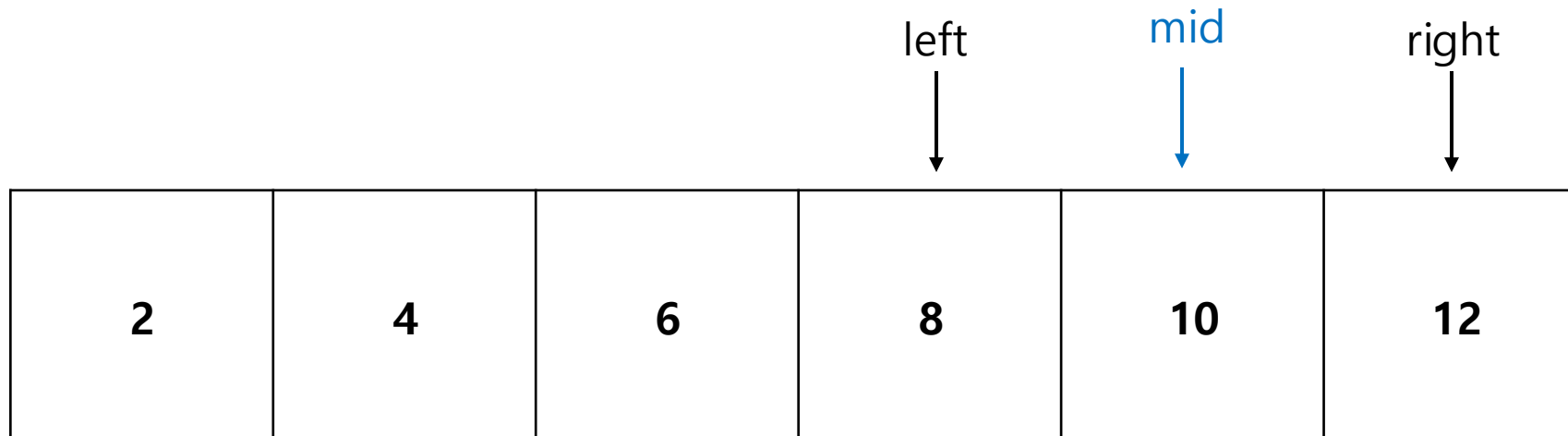
이분탐색 | 01 개요

추가적으로, 만약 배열에 없는 값인 11을 찾는다고 생각해보자.

mid에 위치한 값이 11보다 작다.

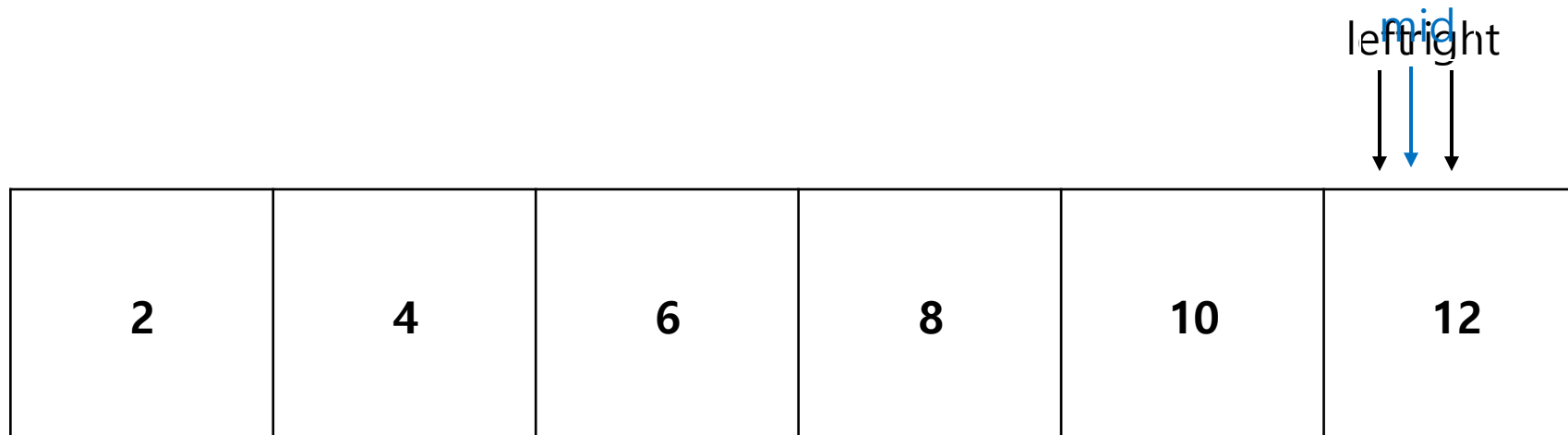
left를 $\text{mid} + 1$ 값으로 둔다.

탐색의 범위는 $[5, 5]$ 가 된다.



이분탐색 | 01 개요

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



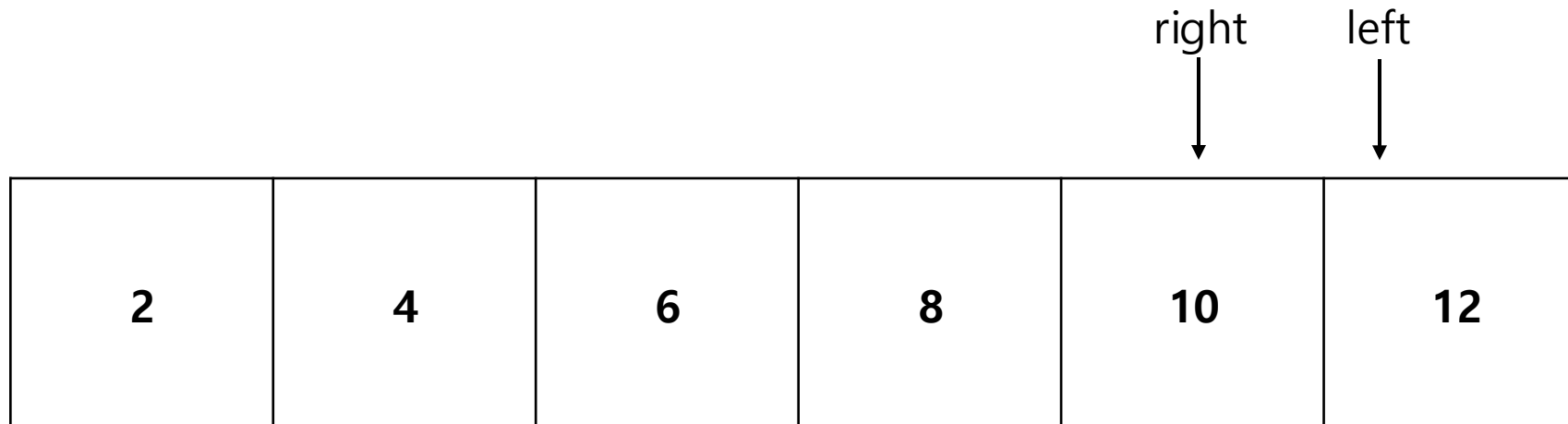
이분탐색 | 01 개요

right를 $\text{mid} - 1$ 값으로 두면, 탐색 범위가 유효하지 않다. (모순이 발생했다는 의미다.)

우리는 구간 $[\text{left}, \text{right}]$ 에서 특정 값이 있는지 여부를 확인하는데, $\text{right} < \text{left}$ 인 상황이 나왔다.

찾고자 하는 값을 못 찾았다는 것이다.

오류의 표시로 -1를 반환하면 된다.



이분 탐색 | 02 구현

- 반복문으로 구현한 이분 탐색

```
5  int binary_search(vector<int> &arr, int target){
6      int left = 0, right = arr.size() - 1;
7
8      while(left <= right){
9          int mid = (left + right) >> 1;
10         if(arr[mid] == target){
11             return mid;
12         }else if(arr[mid] < target){ // target이 더 크면 오른쪽 탐색
13             left = mid + 1;
14         }else{ // target이 더 작으면 왼쪽 탐색
15             right = mid - 1;
16         }
17     }
18     return -1;
19 }
```

이분 탐색 | 02 구현

- 재귀로 구현한 이분탐색

(함수 정의)

`binary_search(arr, left, right, target)` : 배열 `arr`에서 구간 `[left, right]`에 대해 `target`값이 있는지 확인하고 없다면 -1, 있다면 해당 원소의 인덱스를 반환한다고 믿는다(정의한다).

(기저 조건) `left > right`인 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 `target`의 대소관계에 따라 다음을 수행한다.

`arr[mid] > target` 인 경우

배열 `arr`에서 구간 `[left, mid - 1]`에 대해 `target`값이 있는지 확인하고 없다면 -1, 있다면 해당 원소의 인덱스를 반환한다 (작은 문제)

`arr[mid] < target` 인 경우

배열 `arr`에서 구간 `[mid + 1, right]`에 대해 `target`값이 있는지 확인하고 없다면 -1, 있다면 해당 원소의 인덱스를 반환한다 (작은 문제)

이분 탐색 | 02 구현

- 재귀로 구현한 이분 탐색

```
5  int binary_search(vector<int> &arr, int left, int right, int target){
6      if(left > right) return -1;
7
8      int mid = (left + right) >> 1;
9      if(arr[mid] == target) return mid;
10     else if(arr[mid] > target){
11         return binary_search(arr, left, mid - 1, target);
12     }else{
13         return binary_search(arr, mid + 1, right, target);
14     }
15
16 }
```

이분 탐색 | 03 lower_bound, upper_bound

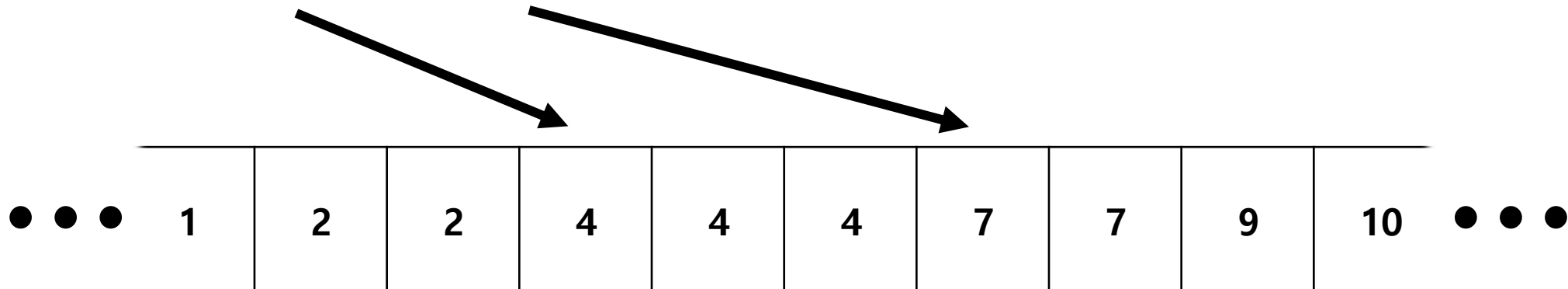
- 개념

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 이상**이 처음 등장하는 위치를 찾는 것이다. (하한)

upper_bound란, 주어진 값 **x 초과**가 처음 등장하는 위치를 찾는 것이다. (상한)

예를 들어 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_bound`와 `std::upper_bound`가 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) $\text{upper_bound}(\text{arr.begin}(), \text{arr.end}(), x) - \text{lower_bound}(\text{arr.begin}(), \text{arr.end}(), x)$ 을 계산해서 값 x가 몇 개 있는지 알 수 있다.

(x인 원소의 개수) = (x 초과인 원소의 위치) - (x 이상인 원소의 위치)

이분 탐색 | 04 예제 (1) : 백준 1920번 수 찾기

- 문제 요약

N 개의 정수 $A[1], A[2], \dots, A[N]$ 이 주어져 있을 때, 이 안에 x 라는 정수가 존재하는지 알아내는 프로그램을 작성하시오

- 입력

- 첫째 줄에 자연수 N 이 주어지고 두번째 줄에는 N 개의 정수 $A[1], A[2], \dots, A[N]$ 이 주어진다.
- 세번째 줄에는 M 이 주어진다. 네번째 줄에는 M 개의 수들이 주어진다.

- 출력

- M 개의 수 각각에 대해 배열 A 에 존재하면 1을, 존재하지 않는다면 0을 출력하라.

이분 탐색 | 04 예제 (1) : 백준 1920번 수 찾기

- 풀이

가장 쉽게 생각할 수 있는 알고리즘은 M 개의 x 에 대해, 배열 A 에 x 이 있는지 선형 탐색하는 것이다.

그러면 시간복잡도는 $O(MN)$ 인데, N 이 최대 $1e5$ 인 점과 M 이 최대 $1e5$ 인 점을 고려하면 $1e10$ (100억)이다. 시간 초과이다.

다른 풀이를 생각해야 한다. 시간복잡도가 $O(N^2)$ 보다 작은 알고리즘을 생각해야 한다.

1. 주어진 A 배열을 정렬한다. $\rightarrow O(N \log N)$
2. M 개의 수에 대해 이분탐색을 진행한다. $O(M * \log N)$

이 전체 시간복잡도는 $O(N \log N)$ 인데, 충분히 통과 가능하다. // $1.6 * 1e6$

정확한 시간 복잡도는 $O((N + M) * \log N)$ 이다.

이분 탐색 | 04 예제 (1) : 백준 1920번 수 찾기

• 구현

- C++ STL인 `binary_search` 함수를 사용하여 구현하면 된다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios::sync_with_stdio(false);
6      cin.tie(nullptr);
7      int N, M;
8      cin >> N;
9      vector<int> A(N);
10     for (int i = 0; i < N; i++) cin >> A[i];
11     sort(A.begin(), A.end());
12     cin >> M;
13     for (int i = 0; i < M; i++) {
14         int X;
15         cin >> X;
16         cout << binary_search(A.begin(), A.end(), X) << "\n";
17     }
18     return 0;
19 }
```

이분 탐색 | 04 예제 (2) : 백준 10816번 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 ($1 \leq N \leq 500,000$)
- 두 번째 줄 : N 개의 숫자카드 (수의 범위 : $-10^7 \sim 10^7$)
- 세 번째 줄 : 찾고 싶은 숫자의 개수 ($1 \leq M \leq 500,000$)
- 네 번째 줄 : 찾고 싶은 숫자 M 개 (수의 범위 : $-10^7 \sim 10^7$)

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

- M 개의 숫자 각각이 숫자 카드에 몇 개 존재하는지 출력한다.

이분 탐색 | 04 예제 (2) : 백준 10816번 숫자 카드 2

- 풀이

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

N, M 의 최대값이 500,000이므로 짜면 당연히 TLE이다.

어떻게 할까?

배열을 정렬해볼까?

예제 입력 1 [복사](#)

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

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

이분 탐색 | 04 예제 (2) : 백준 10816번 숫자 카드 2

- 풀이

쿼리가 들어올 때마다 다음을 수행하면 될 것 같은데?


$(x \text{ 초과인 첫 번째 원소의 위치}) - (x \text{ 이상인 첫 번째 원소의 위치})$

예를 들어, $x = 3$ 의 경우

upper_bound와 lower_bound를 사용하면 되겠다!

예제 입력 1 [복사](#)

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



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

이분 탐색 | 04 예제 (2) : 백준 10816번 숫자 카드 2

- 구현

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int N, M;
5  int main(){
6      ios::sync_with_stdio(false);
7      cin.tie(0);
8
9      cin >> N;
10     vector<int> v(N);
11     for(int i = 0; i < N; i++) cin >> v[i];
12     sort(v.begin(), v.end());
13     cin >> M;
14     int tmp;
15     while(M--){
16         cin >> tmp;
17         cout << upper_bound(v.begin(), v.end(), tmp) - lower_bound(v.begin(), v.end(), tmp) << " ";
18     }
19     return 0;
20 }
```

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 필수이다!

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 K 개의 랜선을 잘라서 N 개의 같은 길이의 랜선을 만들려고 한다. N 개보다 많이 만드는 것도 N 개를 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)

{ K 개의 랜선 길이 값이 차례대로 주어진다.}

- 출력

랜선의 최대 길이를 출력한다.

$$1 \leq K \leq 10,000$$

$$1 \leq N \leq 1,000,000$$

$$1 \leq K \leq 2^{31} - 1$$

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 k 개의 랜선을 잘라서 N 개의 같은 길이의 랜선을 만드는 것이고, 그 길이를 출력하는게 목표이다.

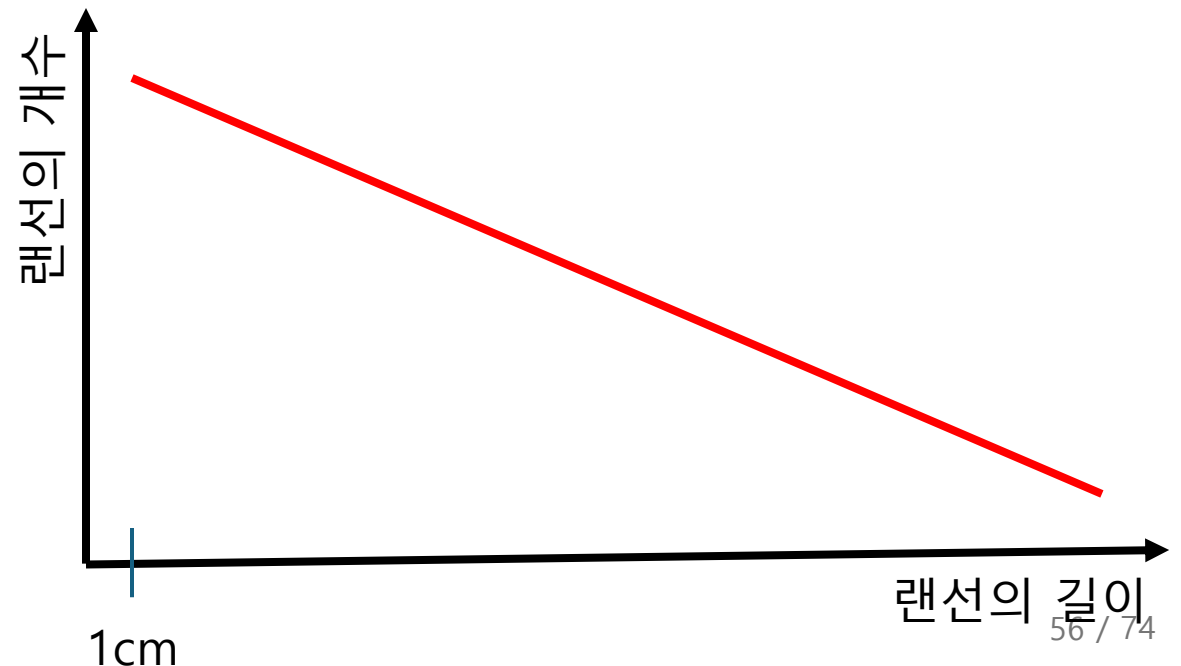
한 가지 떠오르는 방법은 **가장 작은 길이인 1cm** 부터 **k 개의 랜선 중 가장 긴 것의 길이** 까지 잘라보면서 **몇 개의 랜선이 나오는지 확인**하는 것이다.

그걸 함수로 나타내면 이런 '경향'이 나타날 것 같다.

참고) 모든 경우를 탐색할때의 시간복잡도

$O(\text{랜선의 최대 길이} \times \text{랜선의 개수}) \approx 2 * 10^{13}$ 이므로

1초에 1억($1e8$)을 훨씬 초과! **시간 초과!!!**



매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

• 풀이

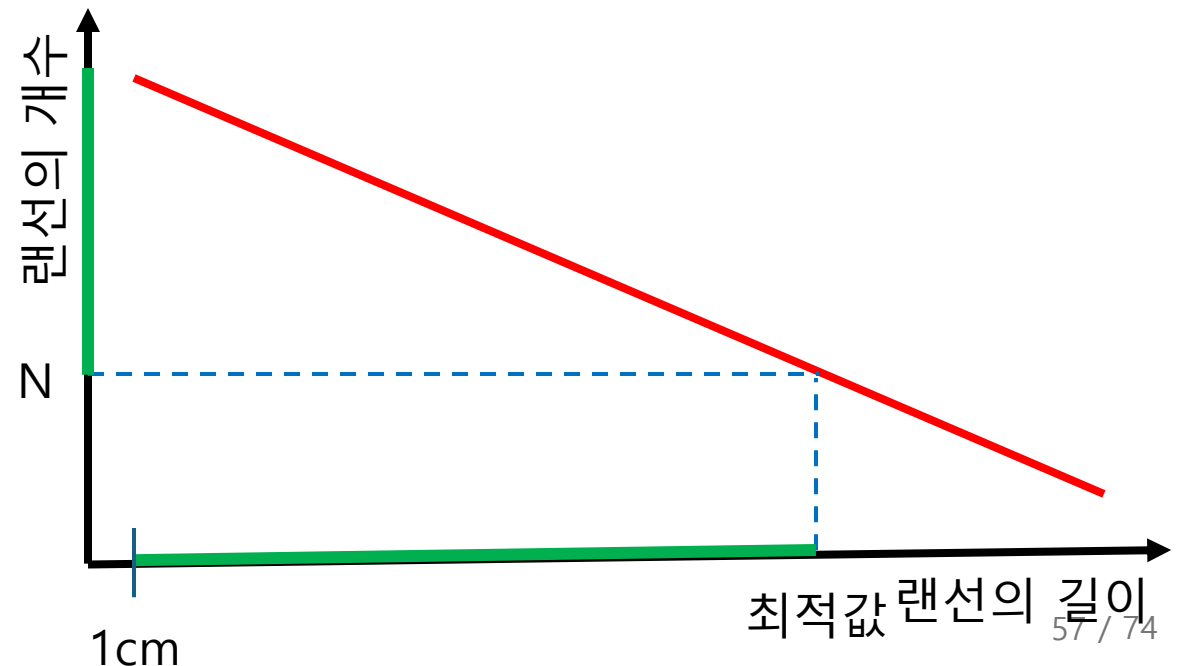
길이가 같은 랜선을 N 개 이상 만들어야 하기 때문에

특정한 값을 기준으로 랜선을 N 개 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.



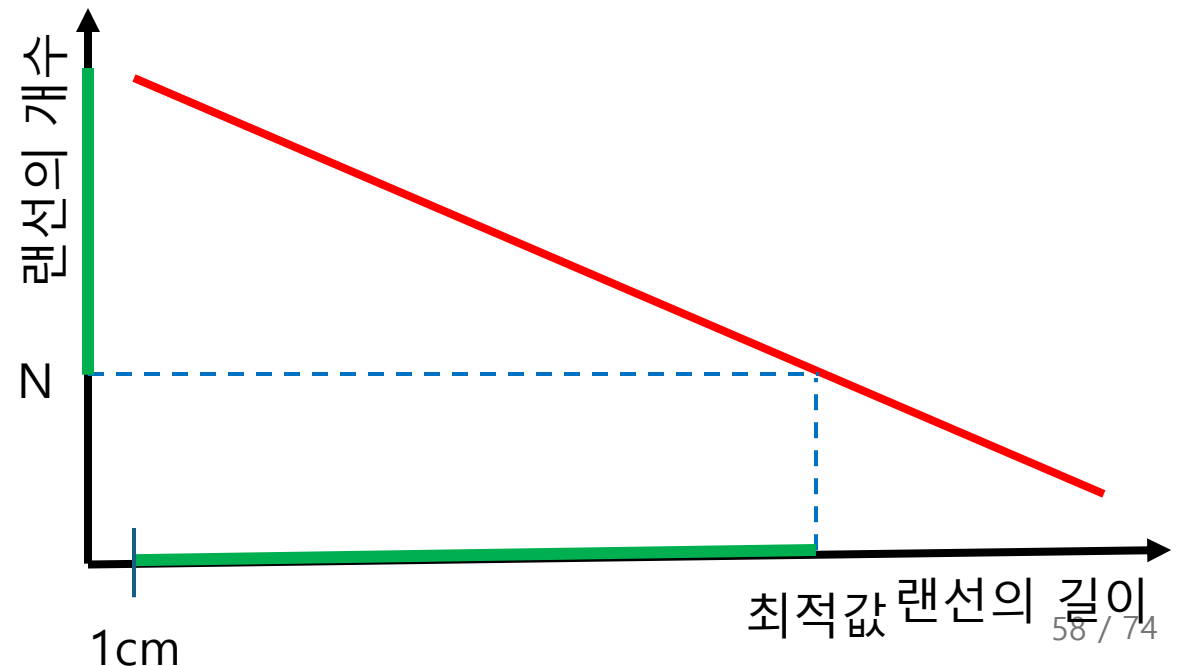
매개변수 탐색 | 02 예제 : 백준 1654번 랜선 자르기

- 풀이

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

굳이 랜선을 모든 길이로 자르지 않아도 된다는 것을 주목해야 한다.

이분탐색의 아이디어를 활용해서 중간 값(길이)에 대해 가능, 불가능 여부에 따라 범위를 조정하면 된다.



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 l로 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 정보

분할 정복 | 01 개요

- 분할 정복(Divide and Conquer)은 문제를 작은 문제로 나누어 해결한 후, 이를 합쳐 전체 문제의 해답을 얻는 기법이다. // 나누어(Divide), 해결한 후(Conquer), 합친다(Combine)!
- 재귀 함수를 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.

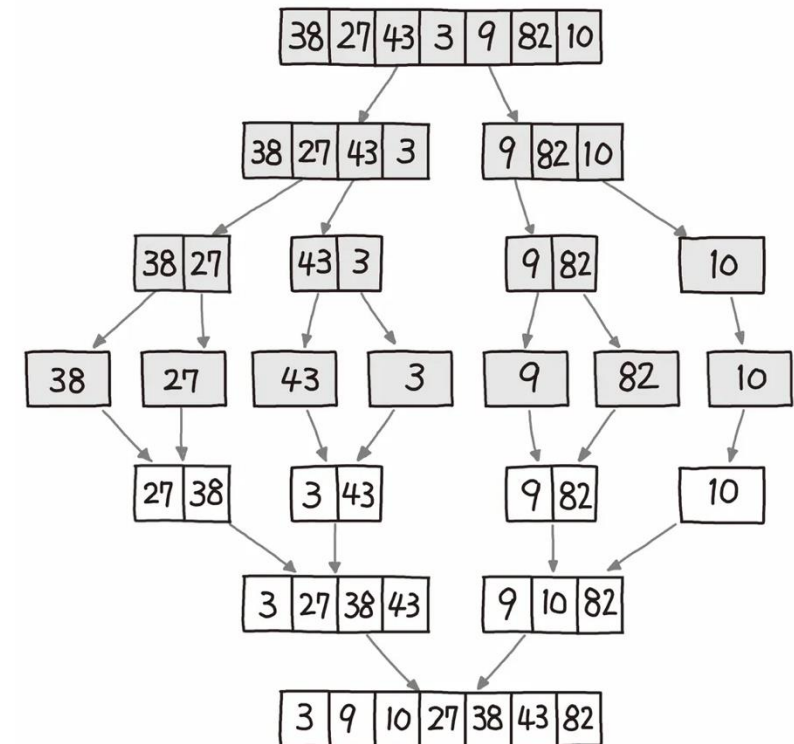
분할 정복 | 02 예제 (1) : 병합 정렬

• 특징

- I. 분할 정복 알고리즘을 기반으로 동작한다.
- II. 배열을 반으로 나누고, 각각을 정렬한 후 합치는 과정을 반복하여 정렬을 수행한다.
- III. 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)

배열을 더 이상 나눌 수 없을 때까지 절반으로 계속 쪼갬다.

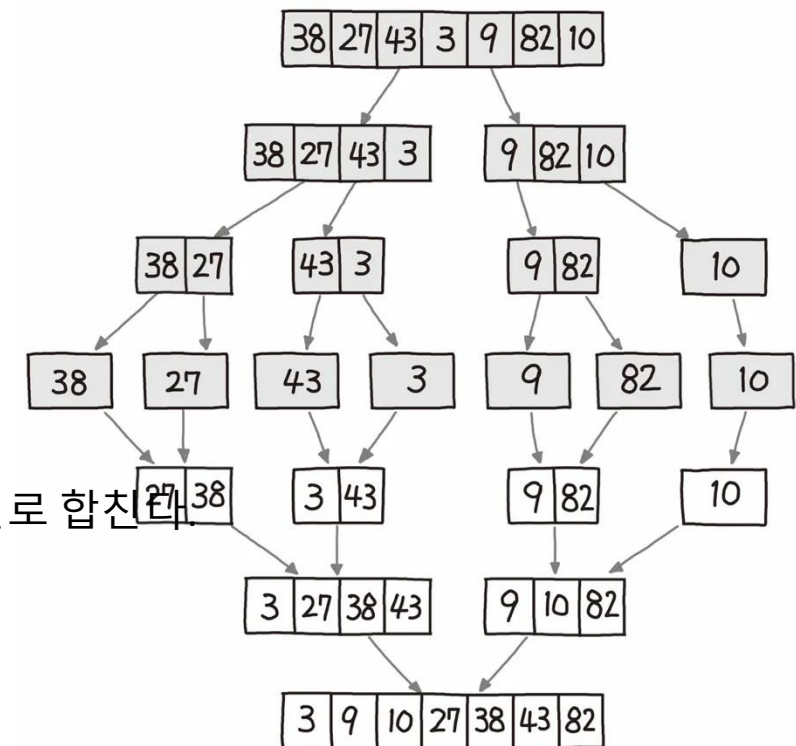
2. 정복 (Conquer)

배열의 크기가 1이면 이미 정렬된 상태이다.

분할된 각각의 부분 배열을 병합하며 정렬한다.

3. 병합 (Merge)

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



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 }
```

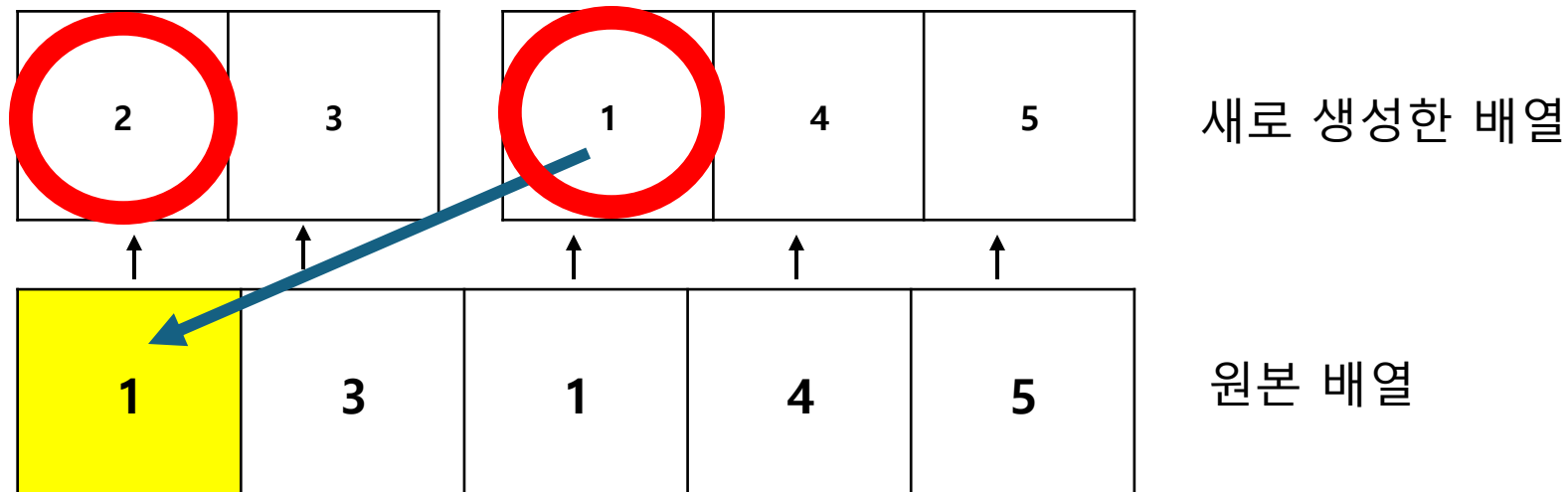
분할 정복 | 02 예제 (1) : 병합 정렬

• 구현

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

새로운 배열 생성하고 원본 배열 값 복사하기.

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

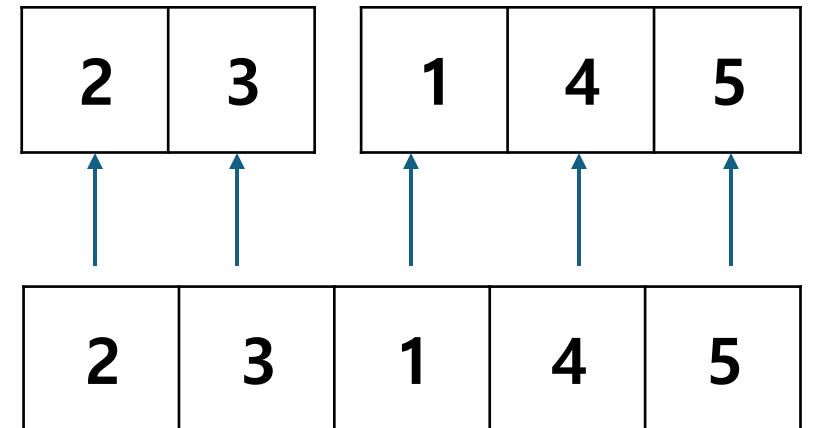


분할 정복 | 02 예제 (1) : 병합 정렬

- 구현

1. 새로운 배열 생성하고 값을 복사하기

```
4 void merge(vector<int> &arr, int left, int mid, int right){
5     int left_size = mid - left + 1; // [left, mid]
6     int right_size = right - mid; // [mid + 1, right]
7
8     vector<int> left_arr(left_size);
9     vector<int> right_arr(right_size);
10
11     // left_arr, right_arr에 데이터 복사
12     for(int i = 0; i < left_size; i++){
13         left_arr[i] = arr[left + i];
14     }
15     for(int j = 0; j < right_size; j++){
16         right_arr[j] = arr[right + j];
17     }
```

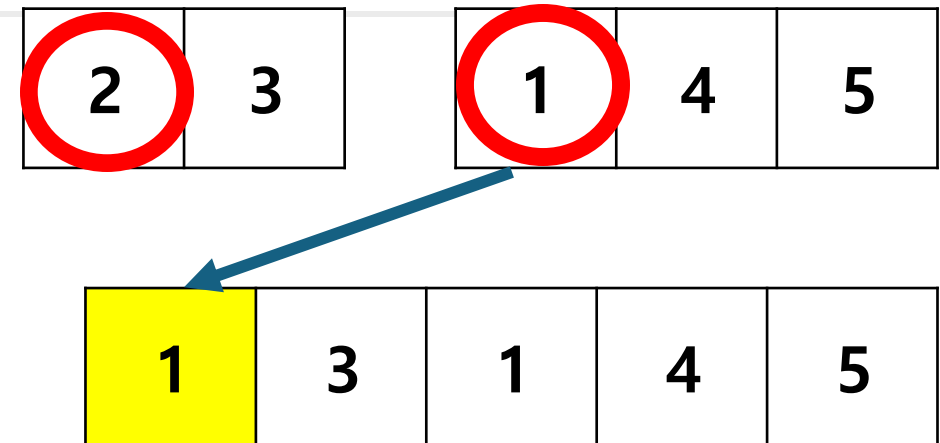


분할 정복 | 02 예제 (1) : 병합 정렬

- 구현

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

```
19 // i : left_arr 인덱스, j : right_arr 인덱스, k : arr 인덱스
20 int i = 0, j = 0, k = left;
21 while(i < left_size && j < right_size){
22     if(left_arr[i] <= right_arr[j]){
23         arr[k++] = left_arr[i++];
24     }else{
25         arr[k++] = right_arr[j++];
26     }
27 }
28
```

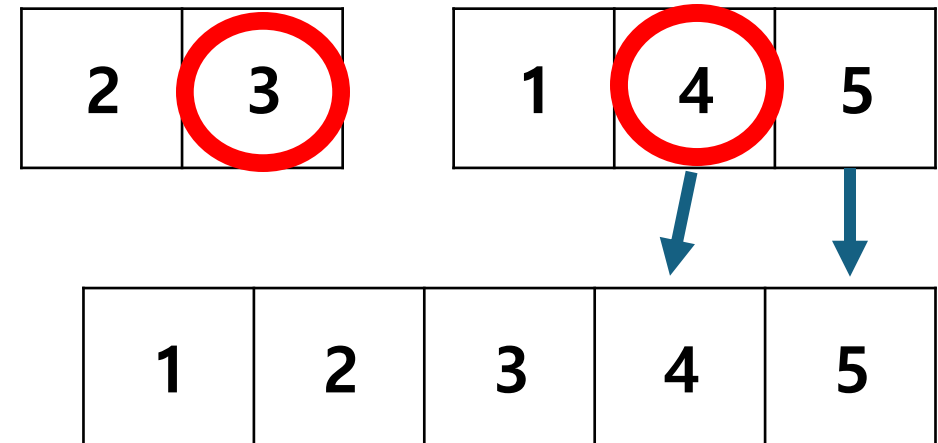


분할 정복 | 02 예제 (1) : 병합 정렬

- 구현

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

```
29      // 남아 있는 요소들 복사
30      while(i < left_size){
31          arr[k++] = left_arr[i++];
32      }
33      while(j < right_size){
34          arr[k++] = right_arr[j++];
35      }
36  }
```



분할 정복 | 02 예제 (2) : 백준 2630번 색종이 만들기

- 문제요약

일정한 규칙에 따라 주어진 $N \times N$ 크기의 종이를 모두 같은 색으로 이루어진 작은 종으로 나누는 문제이다.

- 입력

종이의 한 변의 길이 $N(=2, 4, 8, 16, 32, 64, 128 \text{ 중 하나})$ 이 주어진다. 다음 줄에는 색종이가 주어진다.

하얀색으로 칠해진 칸은 0, 파란색으로 칠해진 칸은 1이다.

- 출력

첫번째 줄에 하얀색 색종이의 개수를 출력하고, 두번째 줄에는 파란색 색종이의 개수를 출력한다.

예제 입력 1 복사

| |
|-----------------|
| 8 |
| 1 1 0 0 0 0 1 1 |
| 1 1 0 0 0 0 1 1 |
| 0 0 0 0 1 1 0 0 |
| 0 0 0 0 1 1 0 0 |
| 1 0 0 0 1 1 1 1 |
| 0 1 0 0 1 1 1 1 |
| 0 0 1 1 1 1 1 1 |
| 0 0 1 1 1 1 1 1 |

예제 출력 1 복사

| |
|---|
| 9 |
| 7 |

분할 정복 | 02 예제 (2) : 백준 2630번 색종이 만들기

- 풀이

- 주어진 색종이가 모두 같은 색으로 칠해져 있는지 확인 (문제) 한다.
- 같은 색으로 칠해져 있지 않으면, 4등분 하여 작은 색종이가 모두 같은 색으로 칠해져 있는지 확인 (작은 문제) 한다.
- 같은 색으로 칠해져 있다면, 하얀색 색종이의 개수 또는 파란색 색종이의 개수를 1 증가해준다.

분할 정복 | 02 예제 (2) : 백준 2630번 색종이 만들기

구현

- func 함수를 다음과 같이 정의한다.

$$\text{Rect} = \{ (x', y') \mid x \leq x' < x + n, y \leq y' < y + n \}$$

Rect에 있는 모든 원소들에 대해 $A[x'][y']$ 의 값이 같으면

해당하는 색종이의 정답배열을 업데이트 해준다.

그렇지 않다면, 색종이를 4등분하고

각각의 왼쪽 위 꼭짓점을 기준으로 재귀호출을 실행한다.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int N, A[212][212], res[2];
5
6  // Rect = {(x', y') | x <= x' < x + n, y <= y' < y + n}
7  void func(int x, int y, int n){
8      int color = A[x][y];
9      for(int i = x; i < x + n; i++){
10         for(int j = y; j < y + n; j++){
11             if(color != A[i][j]){
12                 int half = n >> 1;
13                 func(x, y, half);
14                 func(x + half, y, half);
15                 func(x, y + half, half);
16                 func(x + half, y + half, half);
17             }
18         }
19     }
20 }
21 res[A[x][y]]++;
22 return;
23 }
24
25 int main(){
26     ios::sync_with_stdio(false);
27     cin.tie(0);
28
29     cin >> N;
30     for(int i = 1; i <= N; i++){
31         for(int j = 1; j <= N; j++){
32             cin >> A[i][j];
33         }
34     }
35     func(1, 1, N);
36     cout << res[0] << "\n";
37     cout << res[1] << "\n";
38     return 0;
39 }
```

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

추천 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!

