이분탐색

김영균

1 이분탐색

- **이분 탐색**

  정렬된 배열에서 어떤 수를 $O(\log_2 n)$에 탐색하는 알고리즘

  up&down 숫자 맞추기 게임하는 원리와 비슷하다.

- **예시**

  오름차순으로 정렬된 배열에서 3을 찾아보자

left	mid	right	arr[mid]

index	0	1	2	3	4	5	6	7
arr	-5	-1	0	1	3	5	10	13

1 이분탐색

- lower_bound(first, last, value)

  [first, last)에서 value 이상의 값이 처음으로 나오는 위치를 반환한다.

  만약 찾는 값보다 모두 작다면 마지막 원소 다음 위치 반환

```cpp
vector<int> arr1 = { -5, -1, 0, 1, 1, 2, 2, 2, 4, 5 };
int arr2[] = { -5, -1, 0, 1, 1, 2, 2, 2, 4, 5 };

vector<int>::iterator iter = lower_bound(arr1.begin(), arr1.end(), 1);
int index = iter - arr1.begin();
cout << index << " " << arr1[index] << '\n'; // 3, 1

vector<int>::iterator iter2 = lower_bound(arr1.begin(), arr1.end(), 6);
int index2 = iter2 - arr1.begin();
cout << (iter2 == arr1.end()) << " " << index2 << '\n'; // true, 10

int* iter3 = lower_bound(arr2, arr2 + sizeof(arr2)/sizeof(int), 1);
int index3 = iter3 - arr2;
cout << index3 << " " << *iter3 << '\n'; // 3, 1

int* iter4 = lower_bound(arr2, arr2 + sizeof(arr2)/sizeof(int), 6);
int index4 = iter4 - arr2;
cout << (index4 == 10) << " " << index4; // true, 10
```

1 이분탐색

- upper_bound(first, last, value)

  [first, last)에서 value 보다 큰 값이 처음으로 나오는 위치를 반환한다.

  만약 찾는 값보다 모두 작다면 마지막 원소 다음 위치 반환

```
vector<int> arr1 = { -5, -1, 0, 1, 1, 2, 2, 2, 4, 5 };
int arr2[] = { -5, -1, 0, 1, 1, 2, 2, 2, 4, 5 };

vector<int>::iterator iter = upper_bound(arr1.begin(), arr1.end(), 1);
int index = iter - arr1.begin();
cout << index << " " << arr1[index] << '\n'; // 5, 2

vector<int>::iterator iter2 = upper_bound(arr1.begin(), arr1.end(), 5);
int index2 = iter2 - arr1.begin();
cout << (iter2 == arr1.end()) << " " << index2 << '\n'; // true, 10

int* iter3 = upper_bound(arr2, arr2 + sizeof(arr2)/sizeof(int), 1);
int index3 = iter3 - arr2;
cout << index3 << " " << *iter3 << '\n'; // 5, 2

int* iter4 = upper_bound(arr2, arr2 + sizeof(arr2)/sizeof(int), 5);
int index4 = iter4 - arr2;
cout << (index4 == 10) << " " << index4; // true, 10
```

1 이분탐색

- **예시) 용액**

  지난번 배운 투 포인터를 이용해 풀었던 문제

  투 포인터에서 **왼쪽 포인터를 반복문으로 고정**시키고

  오른쪽 포인터를 이분 탐색으로 찾아도 된다.

  시간복잡도는 $O(n)$에서 $O(nlogn)$ 으로 증가한다.

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

int main() {
    //freopen("input.txt", "r", stdin);
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    int N;
    cin >> N;
    vector<long long> arr(N);
    for(int i = 0; i < N; i++) cin >> arr[i];
    sort(arr.begin(), arr.end());

    int lh, rh;
    long long temp = 0x3f3f3f3f3f3f3f3f;
    for(int i = 0; i < N; i++) {
        int idx = lower_bound(arr.begin() + i + 1, arr.end(), -arr[i]) - arr.begin();

        if(idx != N && arr[i] + arr[idx] == 0ll) {
            lh = i;
            rh = idx;
            break;
        }

        for(int j = -2; j <= 0; j++) {
            if(i < idx + j && idx + j < N && abs(arr[i] + arr[idx + j]) < abs(temp)) {
                temp = arr[i] + arr[idx + j];
                lh = i;
                rh = idx + j;
            }
        }
    }
    cout << arr[lh] << " " << arr[rh];
    return 0;
}
```

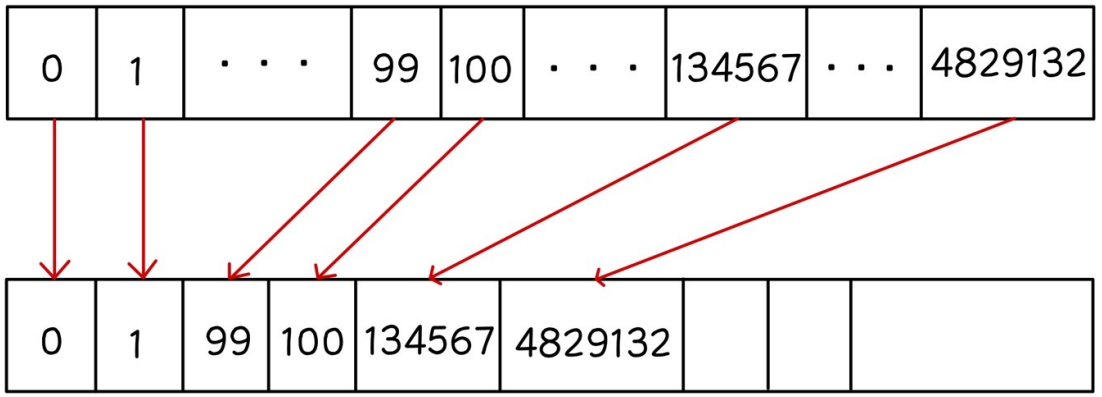
1 이분탐색

- **좌표압축 :** 배열의 값에 새로운 인덱스를 부여하는 기술
  입력으로 주어지는 값의 범위에 비해
  입력의 개수가 턱없이 작을 때 사용할 수 있는 기술

```
bool index[1000000000];
int main() {
    for(int i = 0; i < 1000; i++) {
        int val;
        cin >> val;
        index[val] = true;
    }

    for(int i = 0; i < 1000; i++) {
        int val;
        cin >> val;
        if(index[val]) cout << "배열에 있는 값";
        else cout << "배열에 없는 값";
    }
    return 0;
}
```

→ 메모리 초과




```
vector<int> compression(const vector<int> &arr) {
    vector<int> new_arr = arr;
    sort(new_arr.begin(), new_arr.end());
    new_arr.erase(unique(new_arr.begin(), new_arr.end()), new_arr.end());
    return new_arr;
}
vector<int> pressed = compression(arr);
int search = 123123;
int index = lower_bound(pressed.begin(), pressed.end(), search) - pressed.begin();
```

Q & A

2 파라메트릭 서치

- **파라메트릭 서치란**

  최적화 문제를 결정 문제로 바꾼 후 이분 탐색을 이용해 해결하는 방법

- **최적화 문제란**

  “어떤 최소 값을 구해라”, “어떤 최대 값을 구해라”와 같이 최대 최소 구하는 문제

- **결정 문제란**

  답이 예, 아니오 2개만 나오는 문제

- **예시 - 외판원 문제**

  $optimize(G)$ = 그래프 G의 모든 정점을 한 번씩 방문하고 시작점으로 돌아오는 최단 경로의 길이를 반환한다.

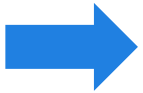
  최적화 문제의 반환 값을 보통 실수나 정수이므로 답의 경우의 수가 무한대이다.

  $decision(G, x)$ = 그래프 G의 모든 정점을 한 번씩 방문하고 시작점으로 돌아오는 길이가 x 이상인 경로가 존재 여부를 반환한다.

  결정 문제의 반환 값은 예 혹은 아니오 이므로 답의 경우의 수가 2개이다.

2 파라메트릭 서치

- **파라메트릭 서치란**
  최적화 문제를 결정 문제로 바꾼 후 이분 탐색을 이용해 해결하는 방법



굳이 바꿔서 풀어야 함??

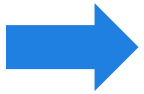
- **결정 문제로 바꿨을 때 이득이 되는 경우**

  1. 최적화 문제를 푸는게 더 어렵다.
  2. 그냥 둘 다 똑같이 어렵다.

- **사실 결정 문제는 최적화 문제보다 어려울 수 없다.**
  최적화 문제를 푸는 방법이 있다면 결정 문제를 무조건 풀 수 있기 때문

```
double optimize(const Graph& G);
bool decision(const Graph& G, double x) {
    return optimize(G) <= x;
}
```

하고자 하는 것



적어도 결정 문제가 최적화 문제보다 더 쉬우니 결정 문제를 해결해
최적화 문제의 답을 구할 수 있는 문제는 결정 문제를 풀어 최적화 문제를 해결하자.

2 파라메트릭 서치

- **파라메트릭 서치로 풀리는 문제**

  결정 문제의 답 구간의 참 또는 거짓이 **한 번** 변하는 문제는 이분 탐색을 이용해 최적화 문제를 해결 할 수 있다.

- **예시 - 나무 자르기**

  상근이는 나무 M미터가 필요하다. 근처에 나무를 구입할 곳이 모두 망해버렸기 때문에, 정부에 벌목 허가를 요청했다. 정부는 상근이네 집 근처의 나무 한 줄에 대한 벌목 허가를 내주었고, 상근이는 새로 구입한 목재절단기를 이용해서 나무를 구할것이다.

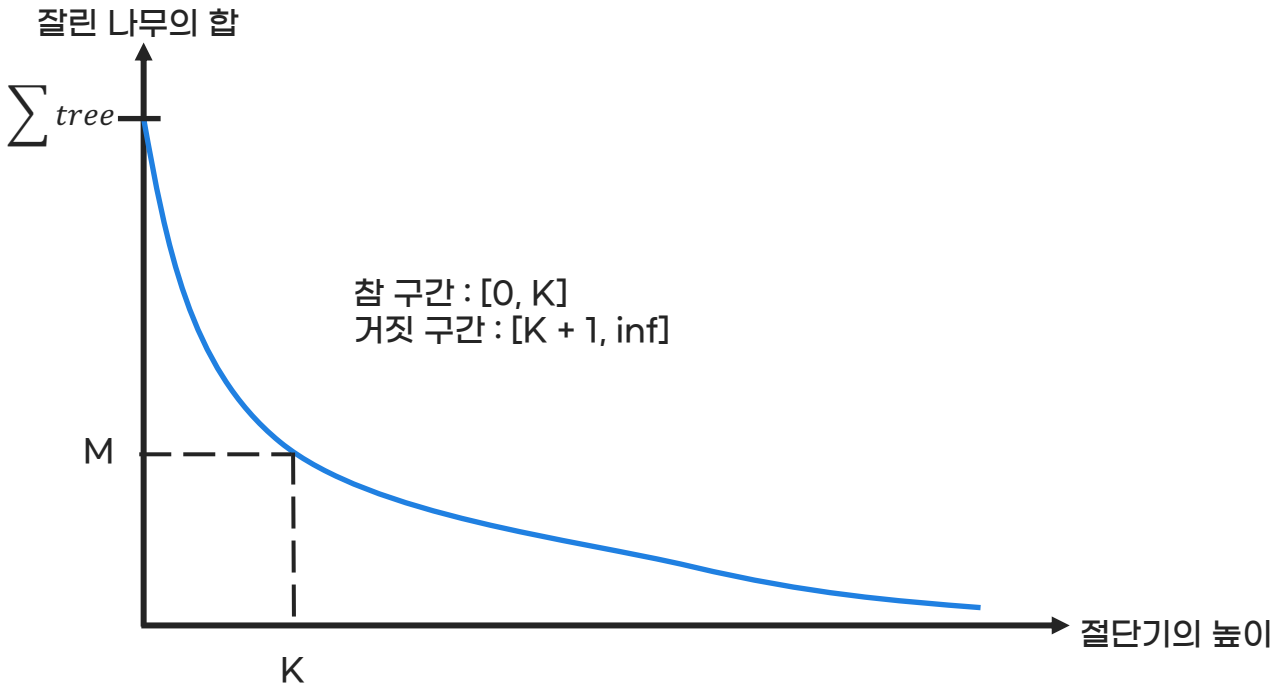
  목재절단기는 다음과 같이 동작한다. 먼저, 상근이는 절단기에 높이 H를 지정해야 한다. 높이를 지정하면 톱날이 땅으로부터 H미터 위로 올라간다. 그 다음, 한 줄에 연속해있는 나무를 모두 절단해버린다. 따라서, 높이가 H보다 큰 나무는 H 위의 부분이 잘릴 것이고, 낮은 나무는 잘리지 않을 것이다. 예를 들어, 한 줄에 연속해있는 나무의 높이가 20, 15, 10, 17이라고 하자. 상근이가 높이를 15로 지정했다면, 나무를 자른 뒤의 높이는 15, 15, 10, 15가 될 것이고, 상근이는 길이가 5인 나무와 2인 나무를 들고 집에 갈 것이다. (총 7미터를 집에 들고 간다) 절단기에 설정할 수 있는 높이는 양의 정수 또는 0이다.

  상근이는 환경에 매우 관심이 많기 때문에, 나무를 필요한 만큼만 집으로 가져가려고 한다. 이때, 적어도 M미터의 나무를 집에 가져가기 위해서 절단기에 설정할 수 있는 높이의 최댓값을 구하는 프로그램을 작성하시오.

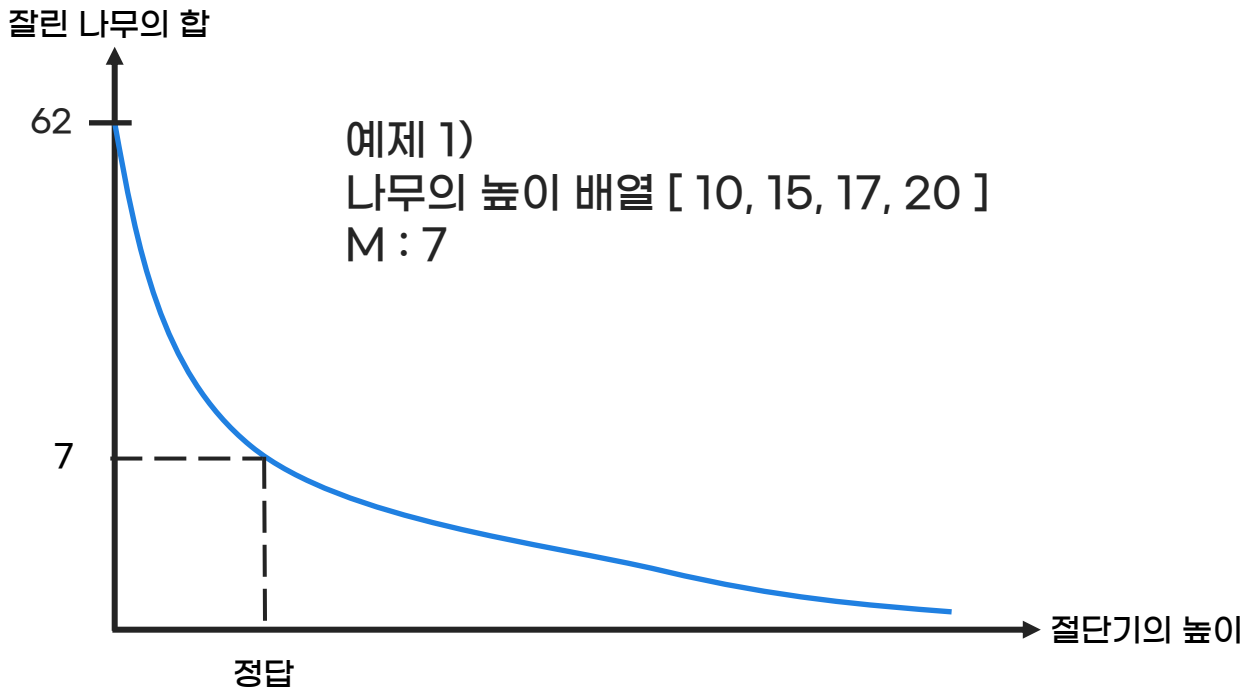
- $optimize(tree, M)$ = 나무의 높이 배열 $tree$가 주어졌을 때 나무를 자른 후 적어도 M미터의 나무를 집에 가져가기 위한 절단기 높이의 최대값을 구해라

2 파라메트릭 서치

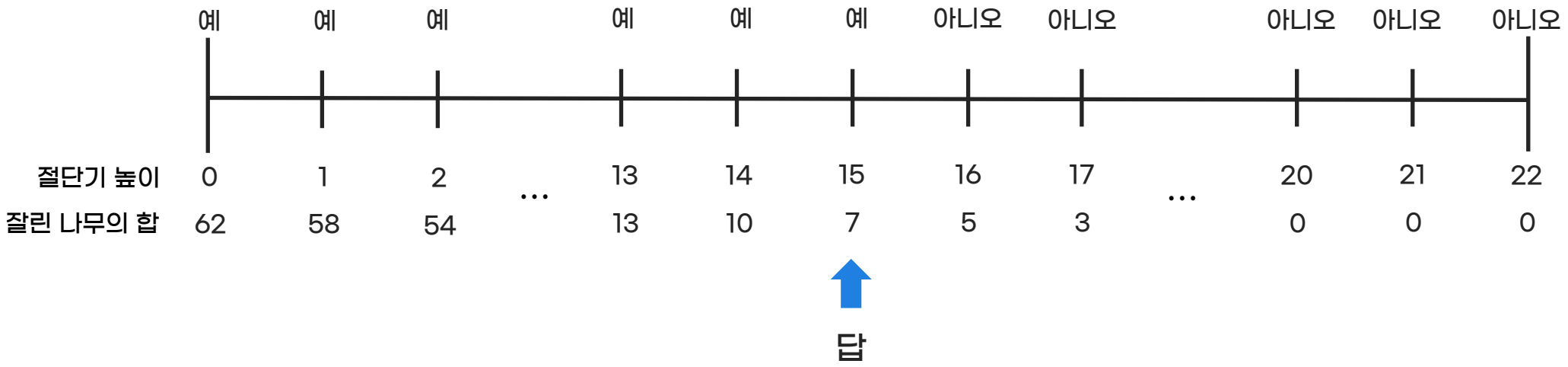
- $optimize(tree, M)$ = 나무의 높이 배열 $tree$가 주어졌을 때 나무를 자른 후
  적어도 M미터의 나무를 집에 가져가기 위한 절단기 높이의 최대값을 구해라
- $decision(tree, M, x)$ = 나무의 높이 배열 $tree$가 주어졌을 때 절단기의 높이를 x로 했을 때 잘린 나무의 높이가 M미터 이상인가?



2 파라메트릭 서치



예제 1)
나무의 높이 배열 [10, 15, 17, 20]
M : 7

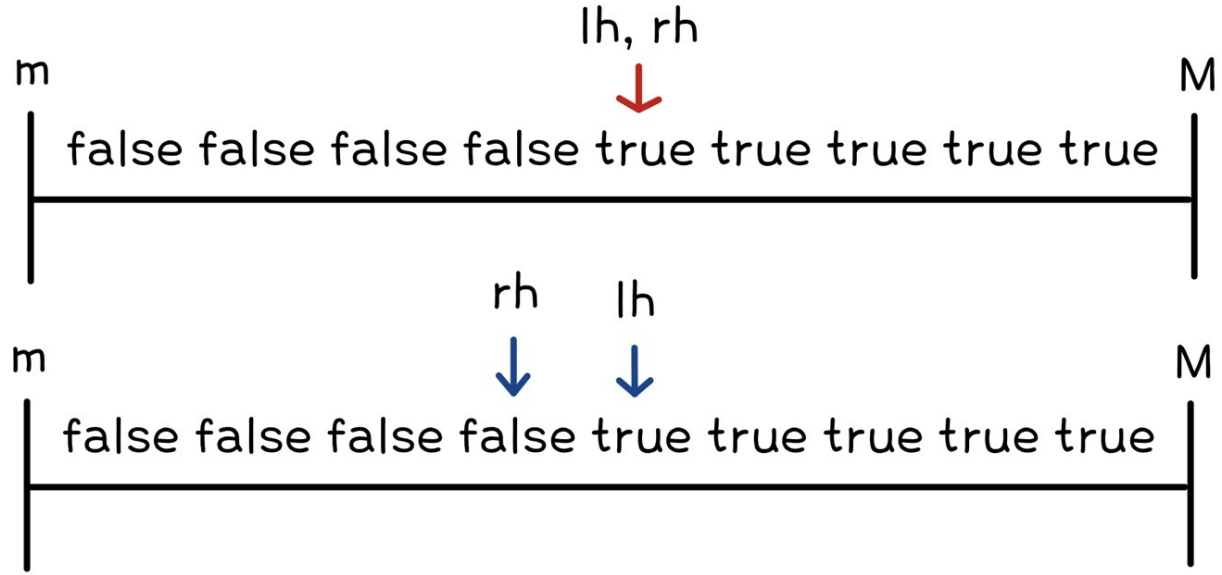


2 파라메트릭 서치

- **답의 구간이 false -> true인 경우**

  $decision(x)$를 만족하는 x의 최솟값을 찾아보자.

  1. 반복문의 탈출 조건, 경계값 lh, rh 설정 -> `while(lh <= rh)`
  2. decision(mid)가 참일 때 `rh = mid - 1`로 이동
  3. decision(mid)가 거짓일 때 `lh = mid + 1`로 이동
  4. while문이 종료되는 경우는 lh와 rh이 교차되는 경우
  5. 답은 `lh`

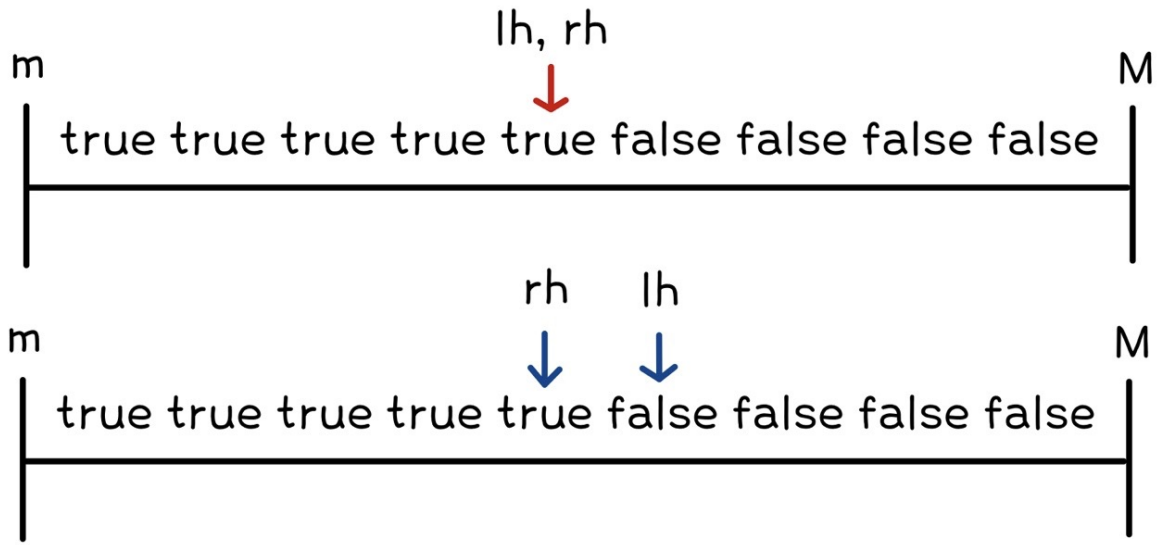


2 파라메트릭 서치

- **답의 구간이 true -> false인 경우**

  $decision(x)$를 만족하는 x의 최대값을 찾아보자.

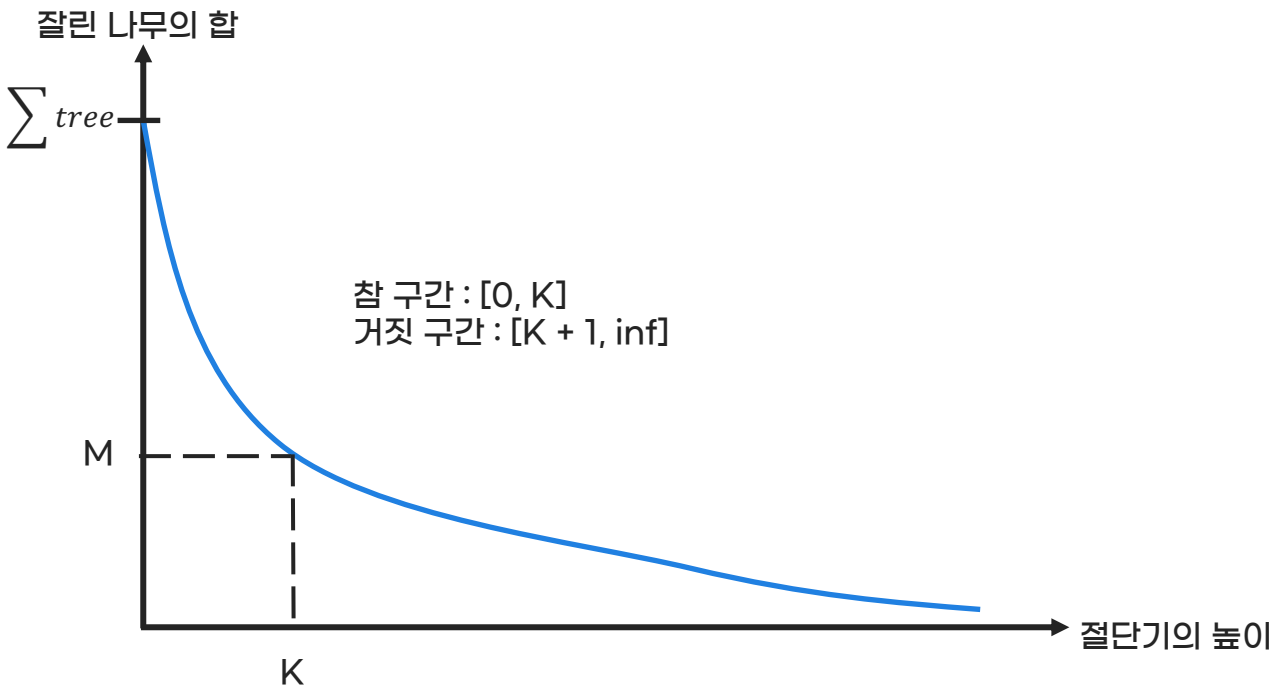
  1. 반복문의 탈출 조건, 경계값 lh, rh 설정 -> `while(lh <= rh)`
  2. decision(mid)가 참일 때 `lh = mid + 1`로 이동
  3. decision(mid)가 거짓일 때 `rh = mid - 1`로 이동
  4. while문이 종료되는 경우는 lh와 rh이 교차되는 경우
  5. 답은 `rh`



2 파라메트릭 서치

- $optimize(tree, M)$ = 나무의 높이 배열 $tree$가 주어졌을 때 나무를 자른 후
  적어도 M미터의 나무를 집에 가져가기 위한 절단기 높이의 최대값을 구해라

- $decision(tree, M, x)$ = 나무의 높이 배열 $tree$가 주어졌을 때 절단기의 높이를 x로 했을 때 잘린 나무의 높이가 M미터 이상인가?




```cpp
#include <bits/stdc++.h>
#define sz(x) ((int)(x).size())
#define all(x) (x).begin(), (x).end()
using namespace std;
using lint = long long;

lint N, M, tree[1000123];
bool decision(int height) {
    lint tot = 0;
    for(int i = 0; i < N; i++) tot += max(0ll, tree[i] - height);
    return tot >= M;
}
int main() {
    //freopen("input.txt", "r", stdin);
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    cin >> N >> M;
    for(int i = 0; i < N; i++) cin >> tree[i];

    int lh = 0, rh = 1e09;
    while(lh <= rh) {
        int mid = (lh + rh) / 2;
        if(decision(mid)) lh = mid + 1;
        else rh = mid - 1;
    }
    cout << rh;
    return 0;
}
```

Q & A