Data Structure

Data Structure

- 컴퓨터에서 데이터를 효율적으로 조회/수정할 수 있도록 저장하는 것을 다루는 분야
- 상황에 따라 적절한 자료구조를 채택해 사용
- 크게 선형 자료구조와 비선형 자료구조로 구분

Data Structure

- Linear: Array, Stack, Queue, Deque, ...
- NonLinear: Tree, Graph, Heap, ...

Data Structure

- 어떠한 명칭(Key)으로 검색해 해당하는 값이 궁금하다: Map
- ~중에 가장 큰 값을 빠르게 찾아야 한다: Heap
- 각 원소들 사이에 관계를 저장해야 한다: Graph
- 구간에 대한 쿼리와 업데이트를 빠르게 하고 싶다: Segment Tree

Data Structure in Class

- Stack
- Queue
- Deque
- List
- Heap
- Tree
  - RedBlack tree
  - Splay tree
- Graph

Data Structure in PS

- Stack -> STL
- Queue -> STL
- Deque -> STL
- List -> STL
- Heap -> STL(Priority Queue)
- Tree
  - RedBlack tree -> STL(Set, Map)
  - Splay tree
- Graph

구현되어 있는 라이브러리를 잘 사용하면 된다

Array

- 여러 개의 변수를 메모리 상에서 연속된 공간에 할당해 사용하는 것
- 선언해둔 배열의 크기는 고정

Vector

- 길이가 늘어나는 배열
- 실제 메모리에 할당되어 있는 용량(Capacity)
- Capacity 중에 현재 사용중인 배열의 크기(Size)
- Size가 Capacity를 넘어가면 메모리를 추가적으로 할당해 사용
  넘지 않는 경우 Capacity에서 사용

STL Vector

- #include <vector>
- std::vector<T> 변수명;

- void push_back(T value) : 벡터 제일 뒤에 value를 추가한다
- void pop_back() : 벡터 제일 뒤에 있는 원소를 제거한다
- T front/back() : 제일 앞/뒤에 있는 원소를 가져온다
- size_type size() : 벡터에 담겨있는 원소의 개수
- T operator[] : 인덱스로 접근

벡터가 비어있다면 RTE

Stack

- Last In First Out(LIFO)
- 마지막에 들어온 값이 먼저 꺼내지는 자료구조

STL Stack

- #include <stack>
- std::stack<value_type> 변수명;

- void push(T value) : 스택에 value를 추가한다
- void pop() : 가장 마지막의 추가한 값을 제거한다
- T top() : 가장 마지막의 추가한 값을 가져온다
- bool empty() : 스택에 담긴 원소가 없다면 true
- size_type size() : 스택에 담겨있는 원소의 개수

스택이 비어있다면 RTE

STL Stack Example

```cpp
#include <iostream>
#include <stack>

using namespace std;

int main() {
    stack<int> st;
    st.push(1);
    st.push(2);
    cout << st.top() << endl;                    // 2
    cout << st.size() << endl;                   // 2
    st.top() = 3;
    cout << st.top() << endl;                    // 3
    st.pop();
    st.pop();
    cout << (st.empty() ? "empty" : "not empty"); // empty;

    return 0;
}
```

Queue

- First In First Out(FIFO)
- 먼저 들어온 값이 먼저 꺼내지는 자료구조(대기열)

STL Queue

- #include <queue>
- std::queue<value_type> 변수명;

- void push(T value) : 큐에 value를 추가한다(enqueue)
- void pop() : 가장 처음에 추가한 값을 제거한다(dequeue)
- T front() : 가장 처음에 추가한 값을 가져온다
- bool empty() : 큐에 담긴 원소가 없다면 true
- size_type size() : 큐에 담겨있는 원소의 개수

큐가 비어있다면 RTE

STL Queue Example

```cpp
#include <iostream>
#include <queue>

using namespace std;

int main() {
    queue<int> q;
    q.push(1);
    q.push(2);
    cout << q.front() << endl;                    // 1
    cout << q.size() << endl;                     // 2
    q.front() = 3;
    cout << q.front() << endl;                    // 3
    q.pop();
    q.pop();
    cout << (q.empty() ? "empty" : "not empty"); // empty;

    return 0;
}
```

Deque

- 삽입과 삭제가 양쪽에서 모두 가능한 자료구조

STL Deque

- #include <deque>
- #include <queue>
- std::deque<value_type> 변수명;

- void push_front(T value) : 가장 앞에 value를 추가한다
- void pop_front() : 가장 앞에 있는 원소를 제거한다
- T front() : 가장 앞에 있는 원소를 가져온다

덱이 비어있다면 RTE

STL Deque

- void push_back(T value) : 가장 뒤에 value를 추가한다
- void pop_back() : 가장 뒤에 있는 원소를 제거한다
- T back() : 가장 뒤에 있는 원소를 가져온다
- bool empty() : 덱에 담긴 원소가 없다면 true
- size_type size() : 덱에 담겨있는 원소의 개수
- void clear() : 덱에 있는 모든 원소를 제거
- T operator[] : 인덱스로 접근

덱이 비어있다면 RTE

STL Deque Example

```cpp
#include <deque>
#include <iostream>

using namespace std;

int main() {
    deque<int> dq;
    dq.push_back(2);
    dq.push_front(1);
    cout << dq.front() << endl;                    // 1
    cout << dq.back() << endl;                     // 2
    cout << dq.size() << endl;                     // 2
    dq.front() = 3;
    cout << dq.front() << endl;                    // 3

    dq.pop_front();
    dq.pop_back();
    cout << (dq.empty() ? "empty" : "not empty"); // empty;

    return 0;
}
```

기본 연습 문제

10828 : 스택

28278 : 스택 2

10845 : 큐

18258 : 큐 2

10866 : 덱

28279 : 덱 2

스택 BOJ 10828

- push X: 정수 X를 스택에 넣는 연산이다.
- pop: 스택에서 가장 위에 있는 정수를 빼고, 그 수를 출력한다. 만약 스택에 들어있는 정수가 없는 경우에는 -1을 출력한다.
- size: 스택에 들어있는 정수의 개수를 출력한다.
- empty: 스택이 비어있으면 1, 아니면 0을 출력한다.
- top: 스택의 가장 위에 있는 정수를 출력한다. 만약 스택에 들어있는 정수가 없는 경우에는 -1을 출력한다.

스택 BOJ 10828

- 기본적인 STL을 이용해 구현하면 된다
- STL을 연습해보자

- 스택 2, 큐, 큐 2, 덱, 덱 2도 모두 마찬가지이다

연습 문제

1406 : 에디터

26043 : 식당 메뉴

1021 : 회전하는 큐

17298 : 오큰수

1725 : 히스토그램

에디터 BOJ 1406

- 일반적인 텍스트 에디터를 만들어보자
- L은 커서를 왼쪽으로 한 칸 이동시키는 것
- R은 커서를 왼쪽으로 한 칸 이동시키는 것
- B는 커서 왼쪽의 문자를 하나 지우는 것
- P $는 커서 왼쪽에 $ 문자를 추가하는 것

에디터 BOJ 1406

- 커서의 왼쪽만 생각해보자.
- 일반적으로 글자는 왼쪽에서 오른쪽으로 쓴다
- 왼쪽에 있는 글자(커서에서 멀리 있는 쪽)는 오래된 글자
- 오른쪽에 있는 글자(커서에서 가까이 있는 쪽)는 최신 글자이다

에디터 BOJ 1406

- 커서를 왼쪽으로 옮긴다면, 가장 마지막에 들어간 데이터를 빼내야 하므로, 스택과 동일하다는 것을 알 수 있다
- 이를 이용해 커서를 왼쪽으로 옮기는 것을 커서 왼쪽에 있는 모든 글자를 스택에 넣어두고 스택에서 빼는 것으로 나타낼 수 있다

에디터 BOJ 1406

- 커서를 왼쪽으로 옮기는 것은 글자 입장에서는 커서의 왼쪽에 있던 글자가 오른쪽으로 이동하는 것
- 앞선 설명에서 이 내용을 스택에서 글자가 빠지는 것으로 구현할 수 있었다
- 반대로 커서를 오른쪽으로 이동하는 것은 제일 마지막에 뺐던 글자를 다시 가져오는 것
- 마지막에 다룬 데이터를 가져오므로 이 또한 스택과 일치한다는 것을 알 수 있다

에디터 BOJ 1406

- 이 둘을 조합하면 커서 기준으로 왼쪽과 오른쪽에 존재하는 글자를 스택으로 각각 관리 하면 커서를 왼쪽과 오른쪽으로 이동하는 것을 관리 할 수 있다

에디터 BOJ 1406

- abcd를 처음에 넣었다 생각하자
- 이는 왼쪽 스택에 a, b, c, d가 들어간 것

에디터 BOJ 1406

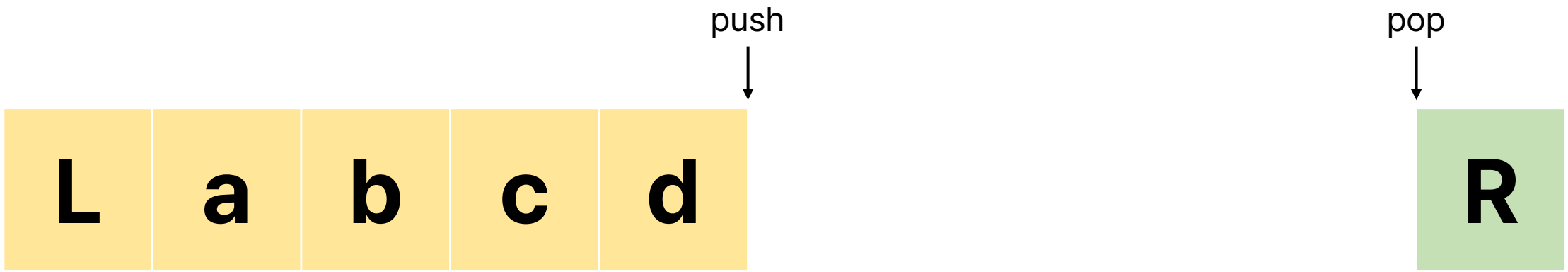
- 커서를 왼쪽으로 옮기는 것은 왼쪽 스택에서 빼서 오른쪽 스택으로 옮기는 것과 동일
- 커서를 오른쪽으로 옮기는 것은 오른쪽 스택에서 빼서 왼쪽 스택으로 옮기는 것과 동일

에디터 BOJ 1406

- 커서 왼쪽 이동

에디터 BOJ 1406

- 커서 오른쪽 이동

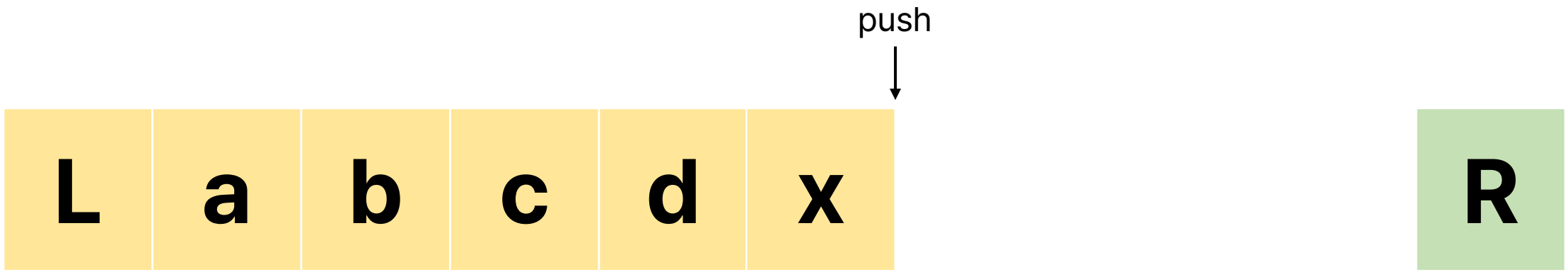


에디터 BOJ 1406

- 커서 오른쪽 이동
- 이동하려 할 때, 빼낼 스택이 비어있다면, 커서가 끝에 도달해 더 이상 움직이지 못하는 것을 의미한다

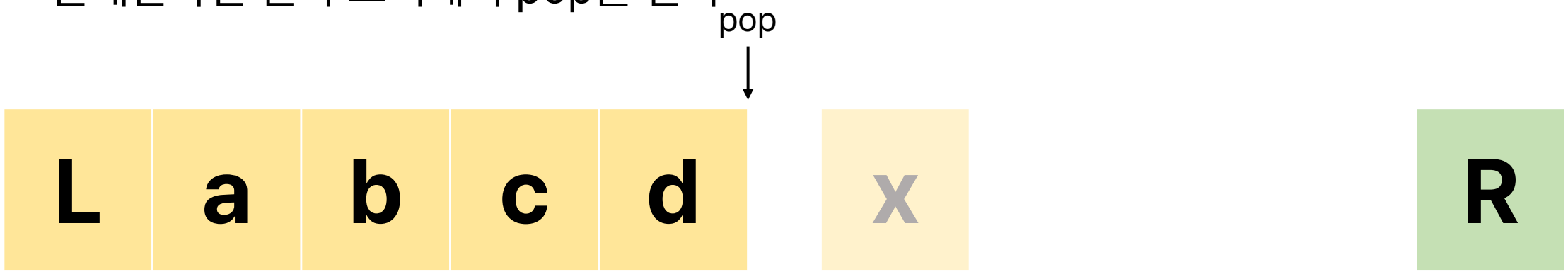
에디터 BOJ 1406

- x 삽입
- 문자를 쓰는 것은 커서 왼쪽에 데이터를 추가하므로 왼쪽 스택에 추가한다



에디터 BOJ 1406

- 문자 하나 제거
- 문자를 하나 제거하는 것은 커서 왼쪽에 있는 문자를 지우므로 왼쪽 스택에 지울 문자가 존재한다면 왼쪽 스택에서 pop을 한다



에디터 BOJ 1406

- 마지막 결과는 모든 글자를 오른쪽 스택으로 넘긴 뒤, 하나씩 pop 하면서 출력한다

식당 메뉴 BOJ 26043

- 식사가 2종류가 있으며, 학생들은 각각 선호하는 식사가 있다
- 선호하는 식사를 한 학생과 선호하지 않은 식사를 한 학생, 식사를 하지 못한 학생을 구분하여 출력하여라

식당 메뉴 BOJ 26043

- 줄을 설 때는 맨 뒤에 줄을 서며, 제일 앞에 있는 학생부터 식사를 하므로 FIFO구조이다
- 즉, 대기열을 큐로 관리하면 된다

- 결과를 출력할 때는 오름차순을 이용해 출력해야 하므로 정렬한 뒤에 출력해야 한다
- 벡터나 배열에 결과를 저장, 정렬 후 출력한다

식당 메뉴 BOJ 26043

- 원하는 식사를 했는지 확인하기 위해, 각 학생이 선호하는 식사를 기억해야 한다
- 배열에 각 학생의 선호 식사를 저장하거나 큐에 넣을 때 학생번호와 선호하는 유형을 같이 넣는다

식당 메뉴 BOJ 26043

- 대기열이 필요 – 큐 1개
- 원하는 식사를 한 학생과 원하는 식사를 하지 못한 학생을 구분하여 저장 – 벡터 2개
- 식사를 하지 못한 학생들을 한 곳에 모아 정렬할 벡터가 필요 – 벡터 1개 또는 기존 벡터 재사용

식당 메뉴 BOJ 26043

- 식사 1인분이 준비됐을 때는 식당 입구에서 대기 중인 학생이 반드시 존재하므로, 대기열에서 학생을 빼기 전에 빈 대기열인지 확인할 필요가 없다

회전하는 큐 BOJ 1021

- 첫 번째 원소를 뽑아낸다
- 왼쪽으로 한 칸 이동한다
- 오른쪽으로 한 칸 이동한다
- 2, 3번 연산을 최소한으로 사용하여 원하는 숫자를 뽑아내라

회전하는 큐 BOJ 1021

- 첫번째(제일 왼쪽)인 경우 2, 3번(회전) 연산을 사용할 필요가 없다
- 왼쪽에 가까운 경우, 왼쪽으로만 이동하는 것이 최소
- 오른쪽에 가까운 경우, 오른쪽으로만 이동하는 것이 최소

- 즉, 한 쪽 방향으로만 이동한다

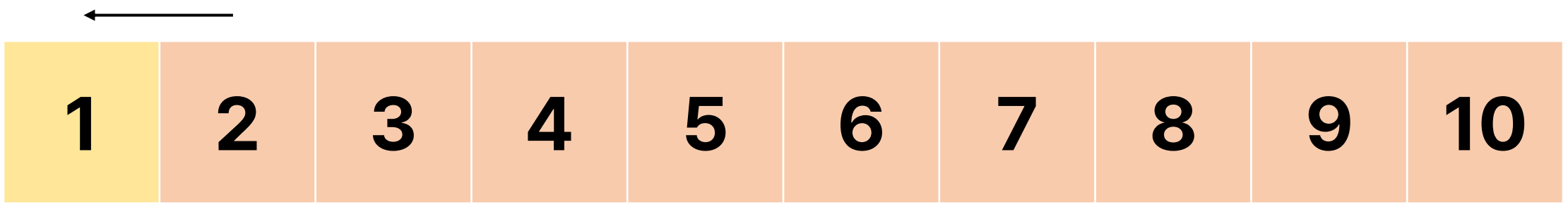
회전하는 큐 BOJ 1021

- 10까지 넣은 큐에서 2를 뽑는다고 해보자

1	2	3	4	5	6	7	8	9	10

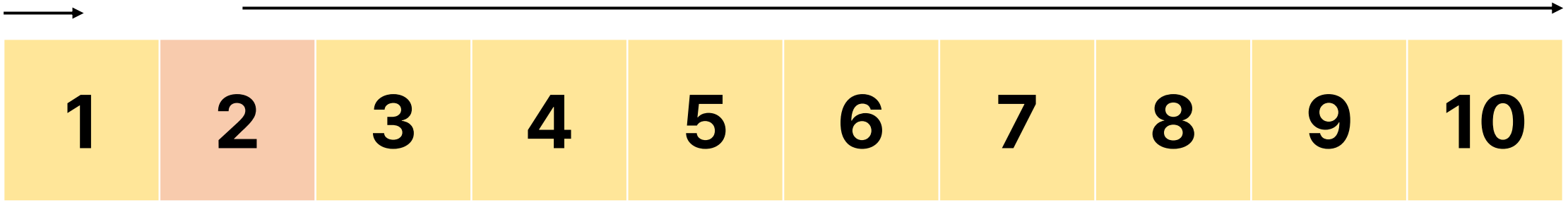
회전하는 큐 BOJ 1021

- 2를 왼쪽으로 이동하는 경우 한 칸만 이동하면 된다



회전하는 큐 BOJ 1021

- 2를 오른쪽으로 이동하는 경우 끝까지 이동하고 한 번 더 이동해야 한다



회전하는 큐 BOJ 1021

- 왼쪽 또는 오른쪽으로 이동하는 자료구조, 특정 원소의 위치를 찾을 수 있는 자료구조가 필요하다
- deque를 이용해 구현이 가능하다
- deque의 find함수를 이용해 원소의 위치를 찾고 더 가까운 방향으로 이동하도록 구현하면 된다

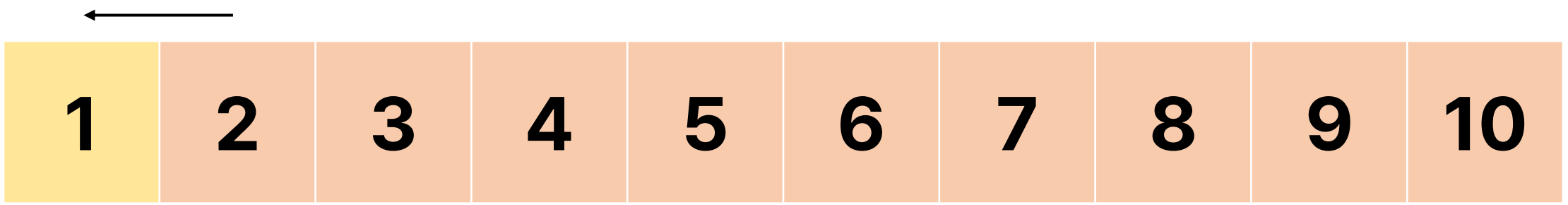
회전하는 큐 BOJ 1021

- 10까지 넣은 큐에서 2를 뽑는다고 해보자

1	2	3	4	5	6	7	8	9	10

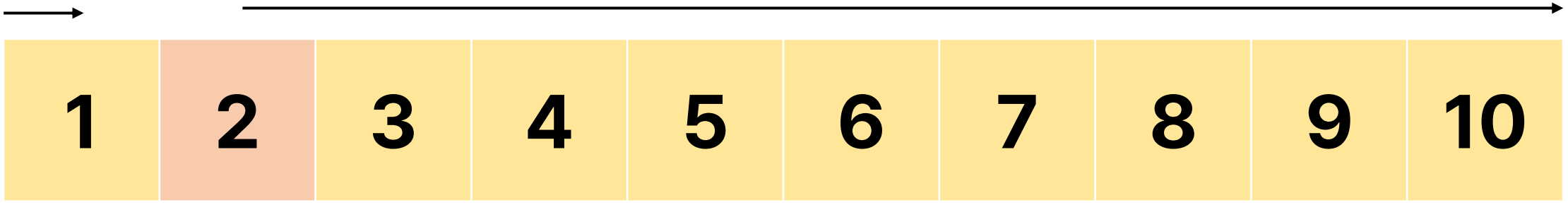
회전하는 큐 BOJ 1021

- 2를 왼쪽으로 이동하는 경우 한 칸만 이동하면 된다



회전하는 큐 BOJ 1021

- 2를 오른쪽으로 이동하는 경우 끝까지 이동하고 한 번 더 이동해야 한다



회전하는 큐 BOJ 1021

- 내가 왼쪽에서 K번째라면
- 왼쪽으로 갈 때는 K – 1번 움직이면 된다
- 오른쪽으로 갈 때는 N – K + 1번 움직이면 된다

- 둘을 합치면 N임을 알 수 있다

회전하는 큐 BOJ 1021

- 뽑으려는 수가 첫번째에 올 때까지 이동시킨다
- 그렇다면 반대쪽으로 이동할 때 사용할 연산의 횟수는 N-(한쪽 이동에 사용한 연산수)이다

- 둘 중 최소값을 더하면 된다

Monotone Stack

- 스택의 값들을 강한 증가/감소 또는 단조 증가/감소하게 관리하는 것

- 강한 증가: $a[i] < a[i+1]$
- 강한 감소: $a[i] > a[i+1]$
- 단조 증가: $a[i] \leq a[i+1]$
- 단조 감소: $a[i] \geq a[i+1]$

Monotone Stack

- 스택의 값을 넣을 때 강한 증가/감소 또는 단조 증가/감소 조건에 어긋나는 원소들을 모두 꺼낸 후 값을 넣는다

- 감소 스택에 9, 7, 5, 3이 들어있을 때(stack = {9, 7, 5, 3}) 6을 넣는다면
- 3을 제거
- 5를 제거
- 6을 삽입
- 스택에는 9, 7, 6이 남는다(stack = {9, 7, 6})

오큰수 BOJ 17298

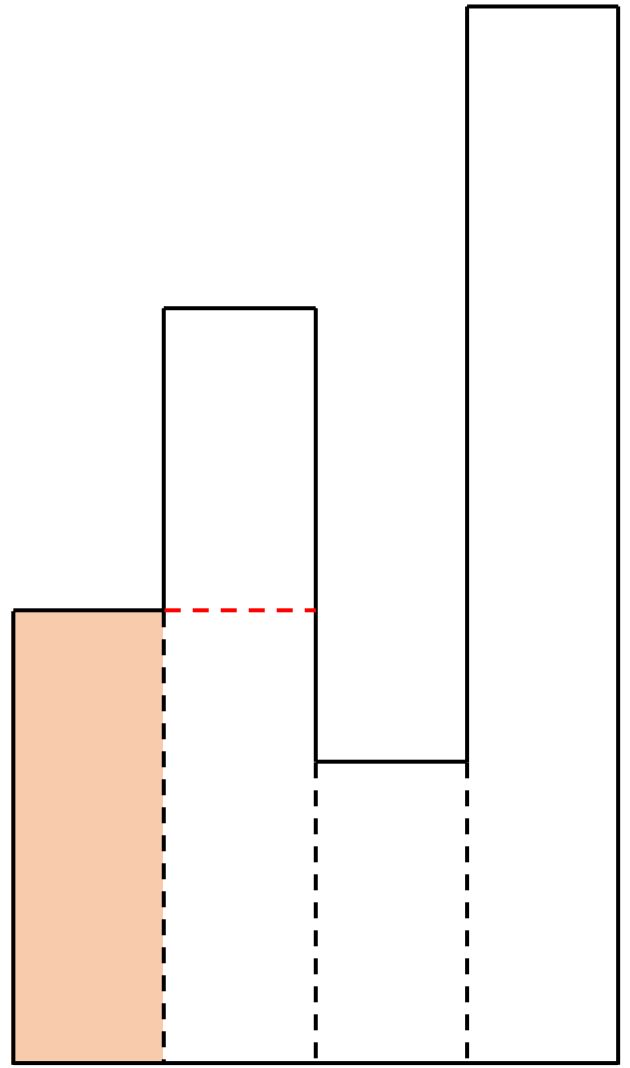
- 수열 $A = A_1, A_2, \ldots, A_N$가 주어진다
- 오큰수는 A_i보다 오른쪽에 있으며 A_i보다 큰 수 중에서 가장 왼쪽에 있는 수이다
- 오큰수가 존재하지 않는 경우 오큰수는 -1이다

오큰수 BOJ 17298

- 예제 1

  4

  3 5 2 7

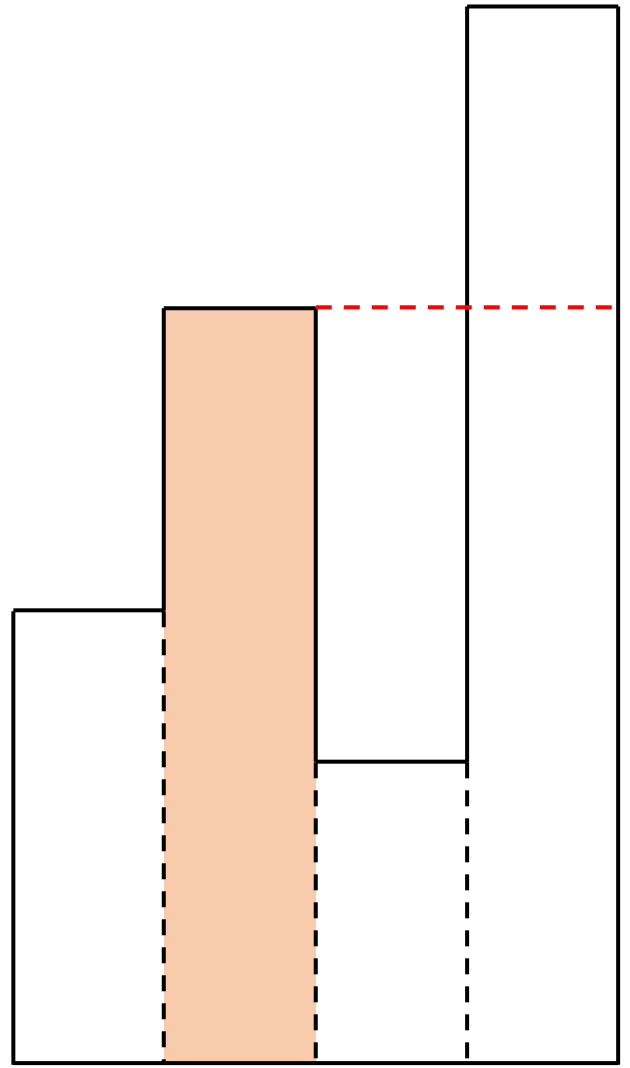


오큰수 BOJ 17298

- 예제 1

  4

  3 **5** 2 7

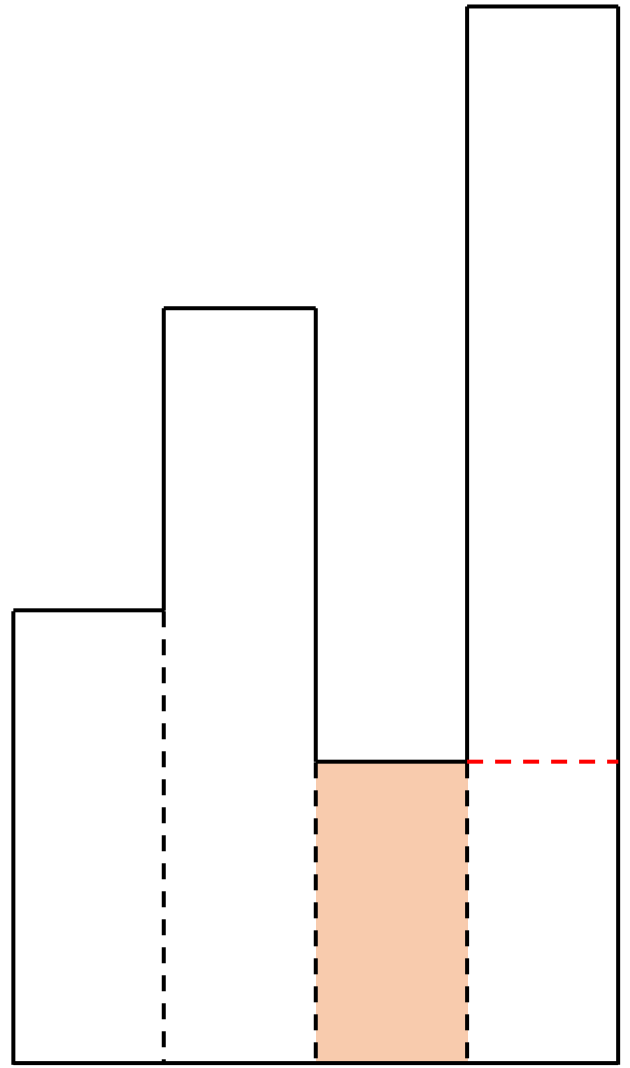


오큰수 BOJ 17298

- 예제 1

  4

  3 5 **2** 7

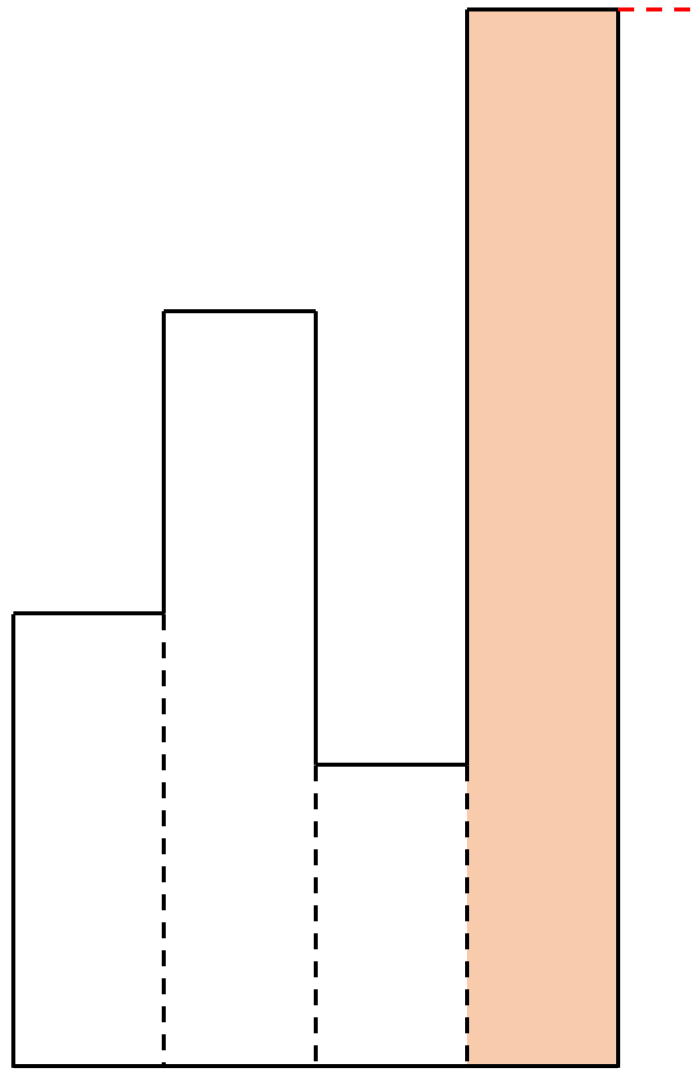


오큰수 BOJ 17298

- 예제 1

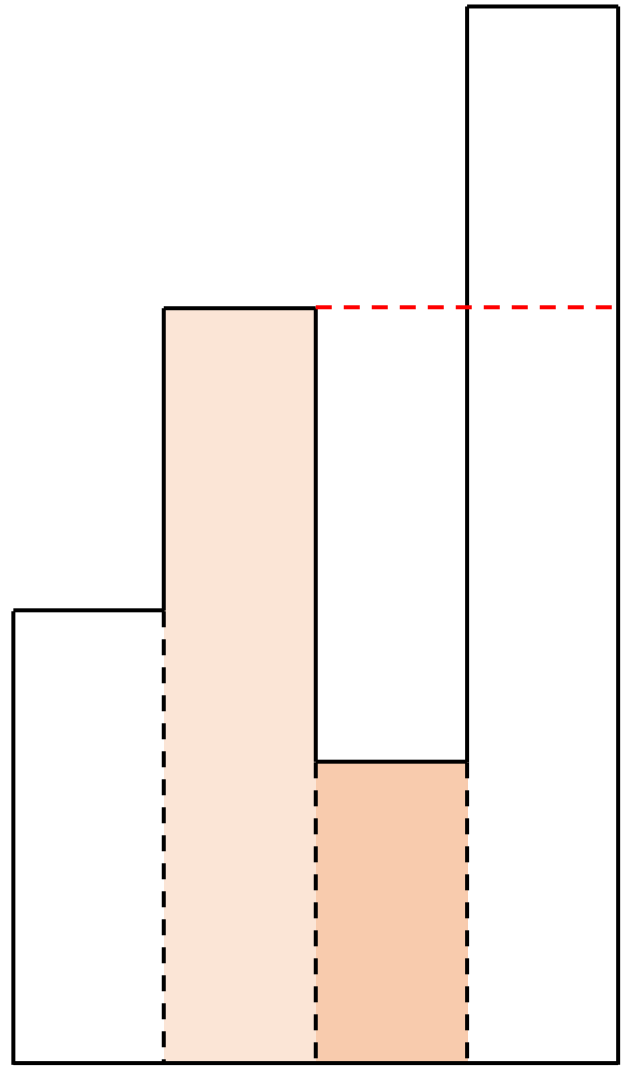
  4

  3 5 2 **7**



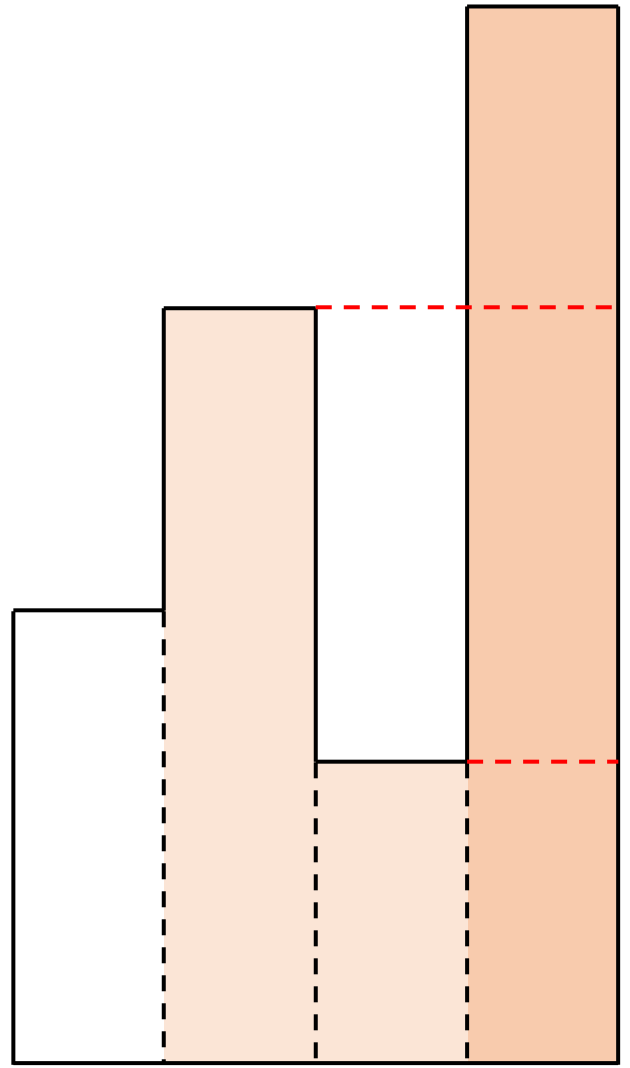
오큰수 BOJ 17298

- 어떠한 자료구조를 사용해 아직 오큰수가 정해지지 않은 수를 저장한다고 하자
- 3을 보고 있을 때는 5는 아직 오큰수가 정해지지 않았다
- 현재 A_i를 보고 있을 때 $A_j (j < i)$의 오큰수가 정해지지 않았다면 A_j이후로 계속 감소하는 숫자만 나왔다는 뜻이다



오큰수 BOJ 17298

- 현재 자료구조에는 5와 3이 저장되어 있다
- 7이 나오는 순간 5와 3의 오큰수가 정해지게 된다
- 자료구조에는 계속 감소하는 순서대로 값이 저장되며 마지막에 들어간 수가 더 작으므로 마지막 수부터 꺼내보며 오큰수가 어디까지 정해졌는지 살펴본다
- 즉, 모노톤 스택을 사용하면 문제를 해결할 수 있다

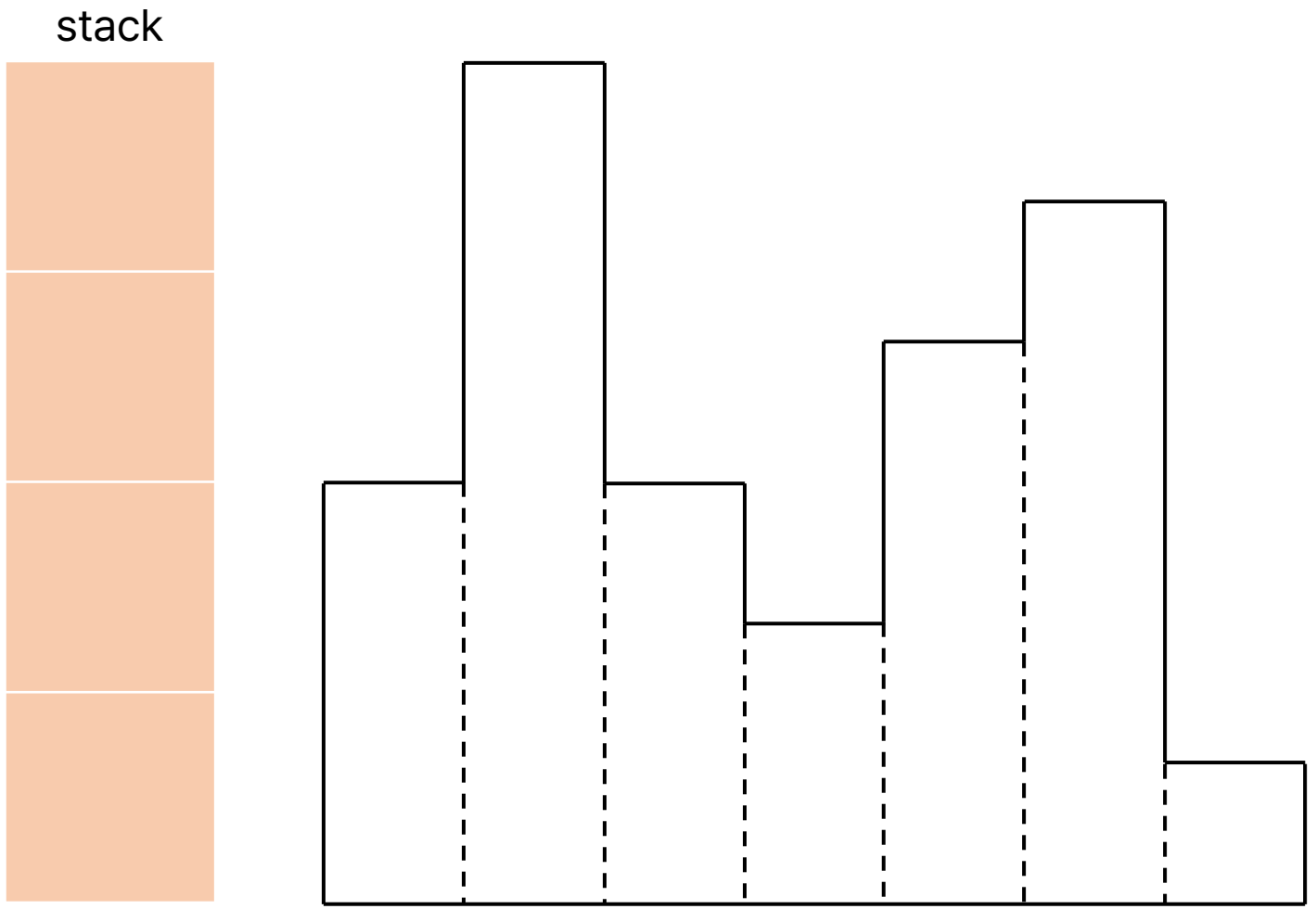


오큰수 BOJ 17298

- 더 큰 예제

  7

  3 6 3 2 4 5 1

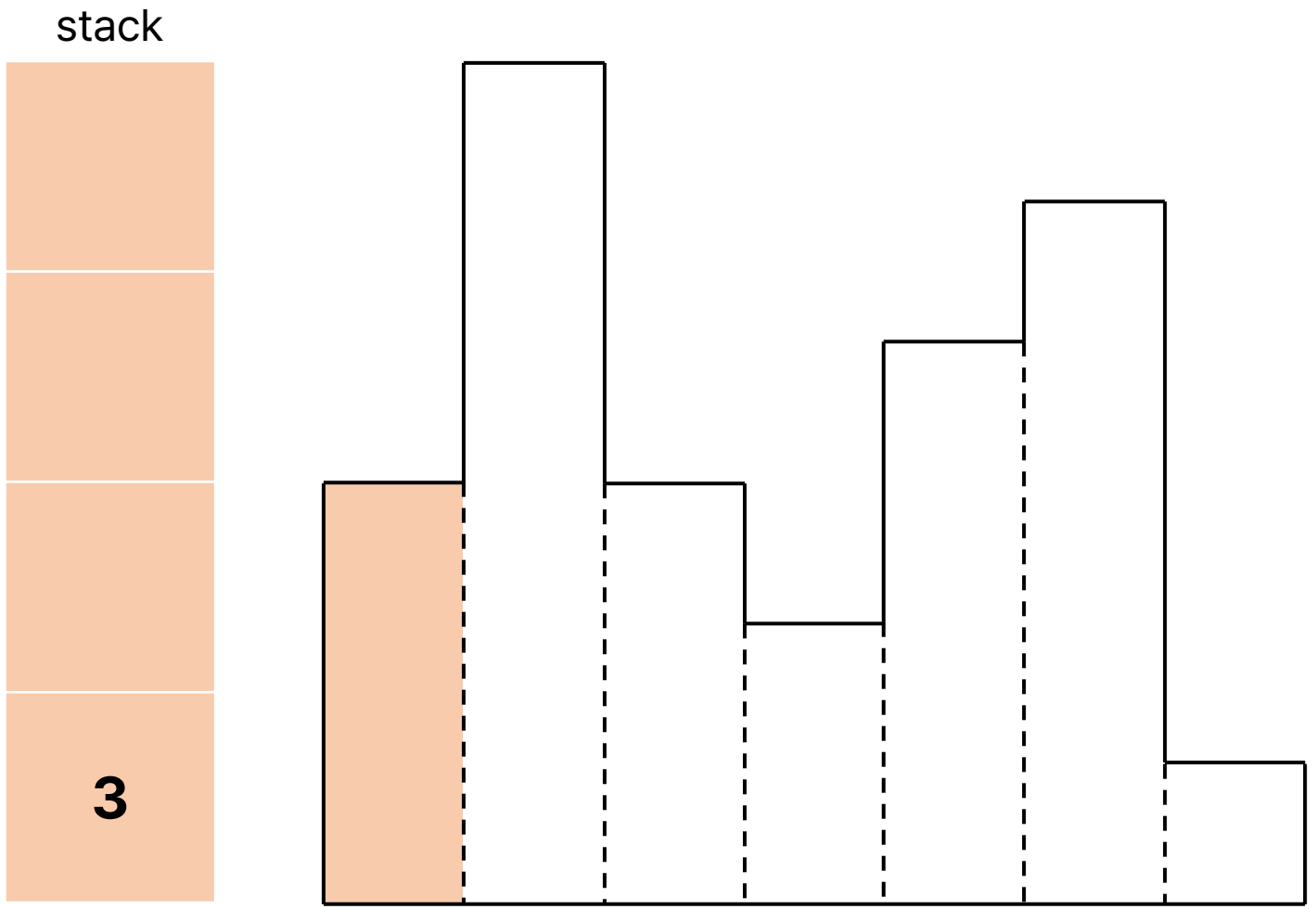


오큰수 BOJ 17298

- 더 큰 예제

  7

  3 6 3 2 4 5 1



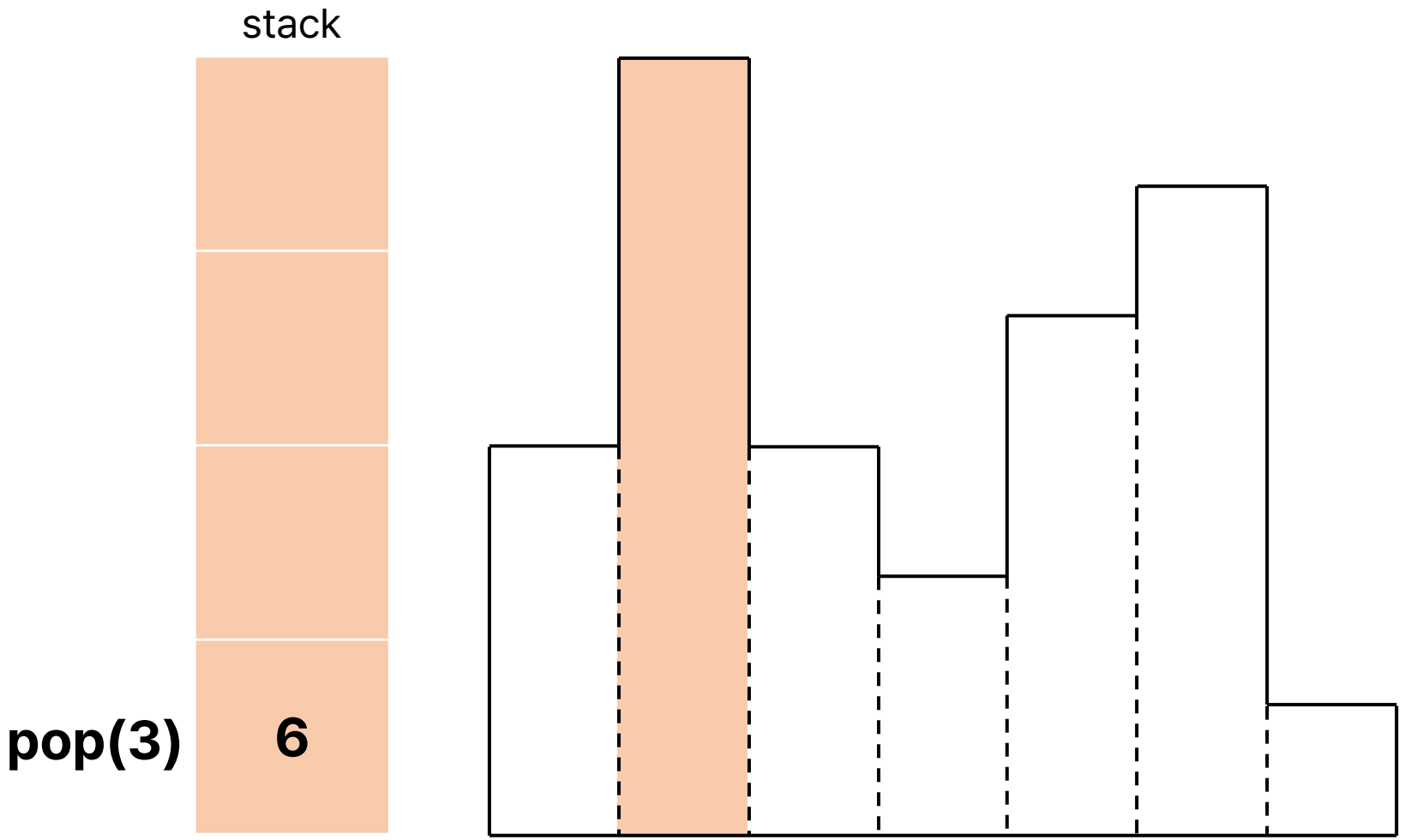
오큰수 BOJ 17298

- 더 큰 예제

  7

  3 **6** 3 2 4 5 1

  6



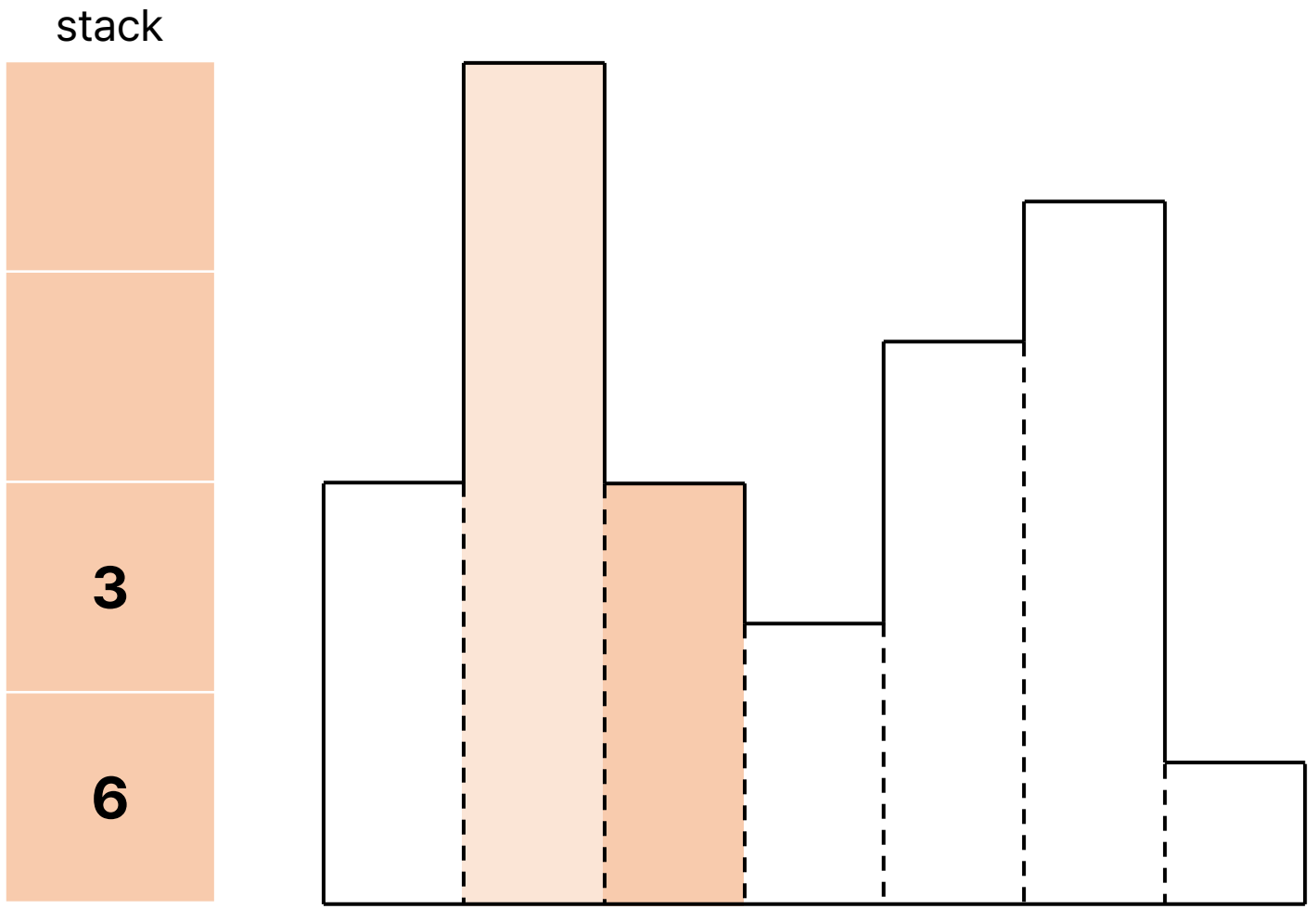
오큰수 BOJ 17298

- 더 큰 예제

  7

  3 6 **3** 2 4 5 1

  6



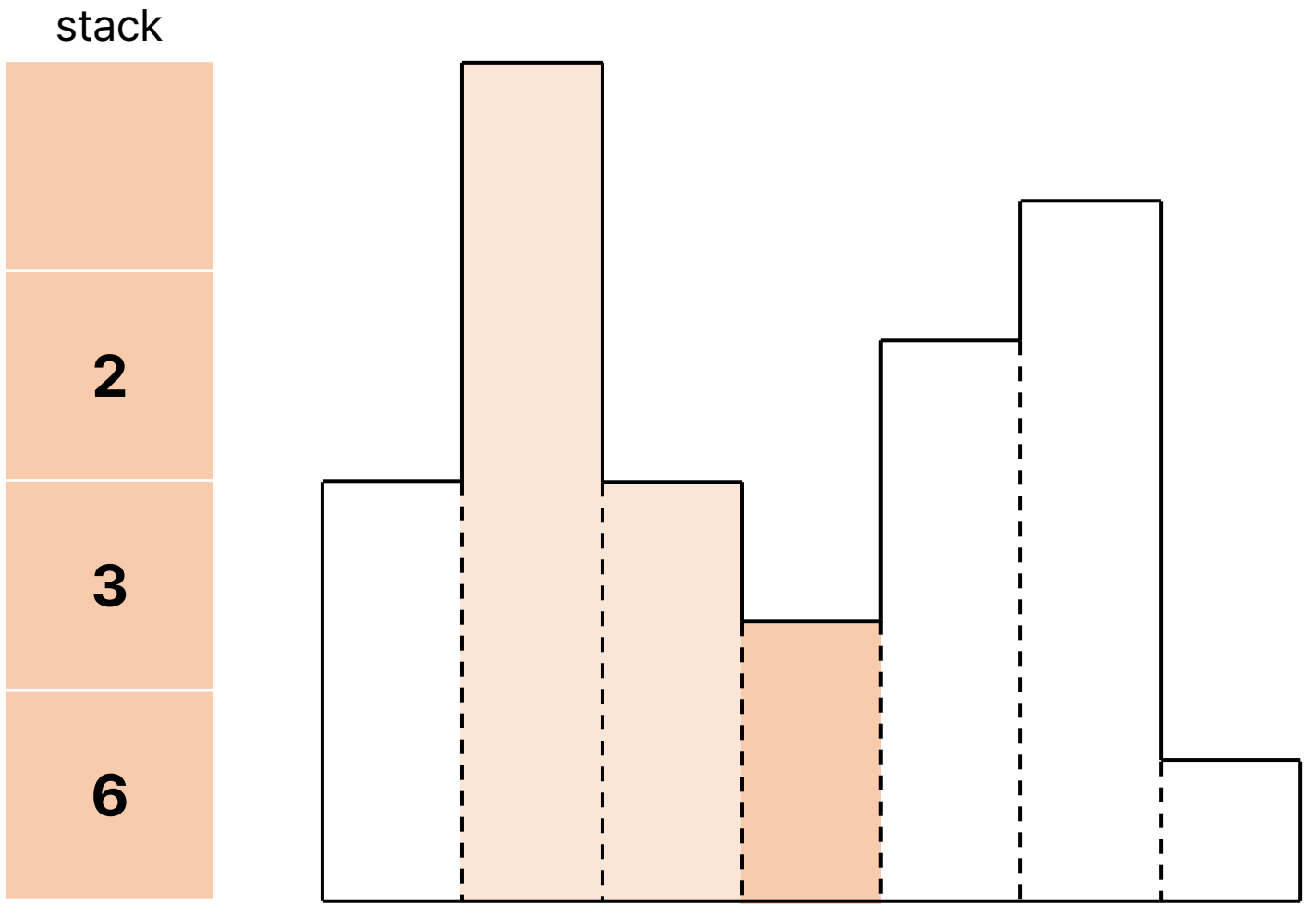
오큰수 BOJ 17298

- 더 큰 예제

  7

  3 6 3 **2** 4 5 1

  6



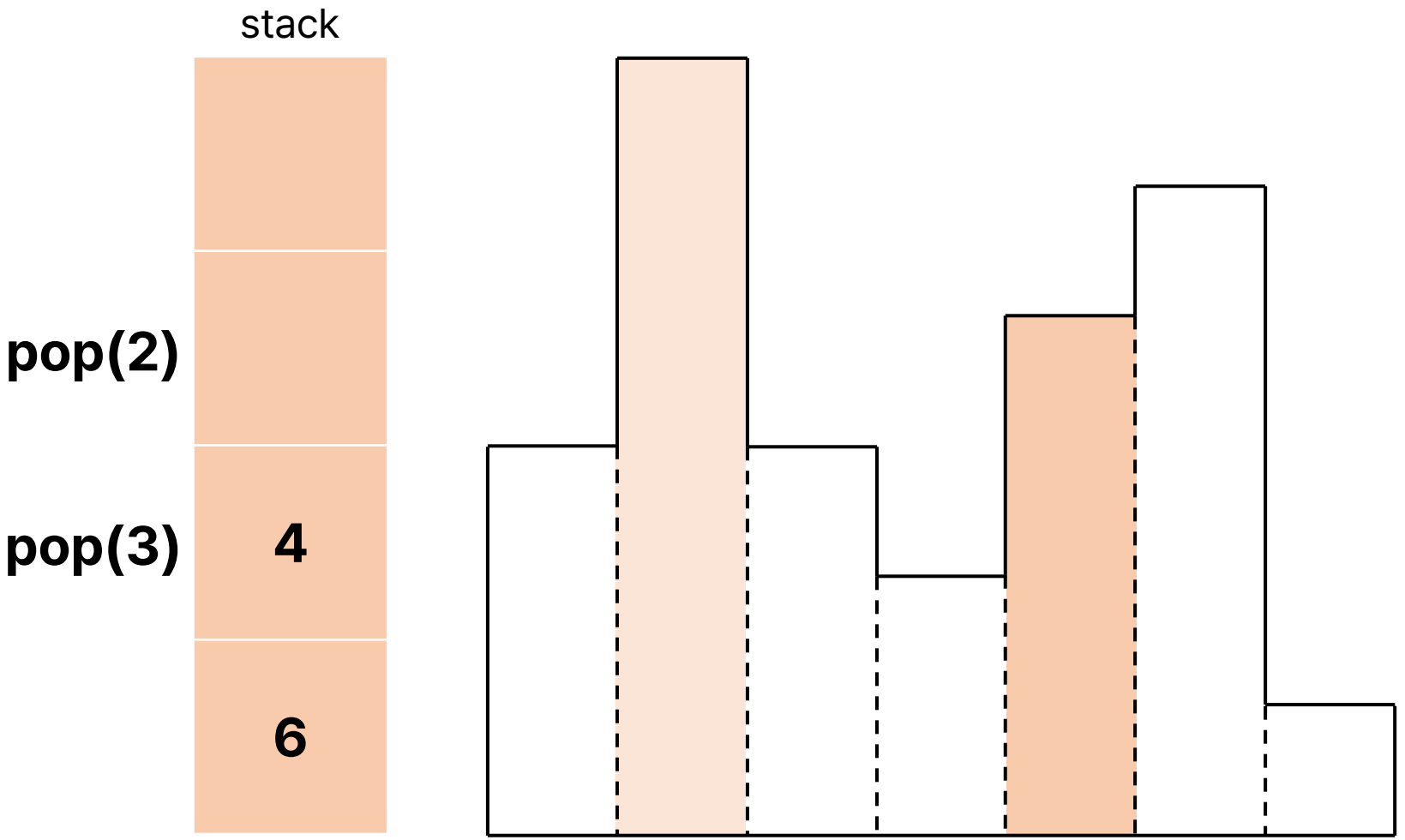
오큰수 BOJ 17298

- 더 큰 예제

  7

  3 6 3 2 **4** 5 1

  6 4 4



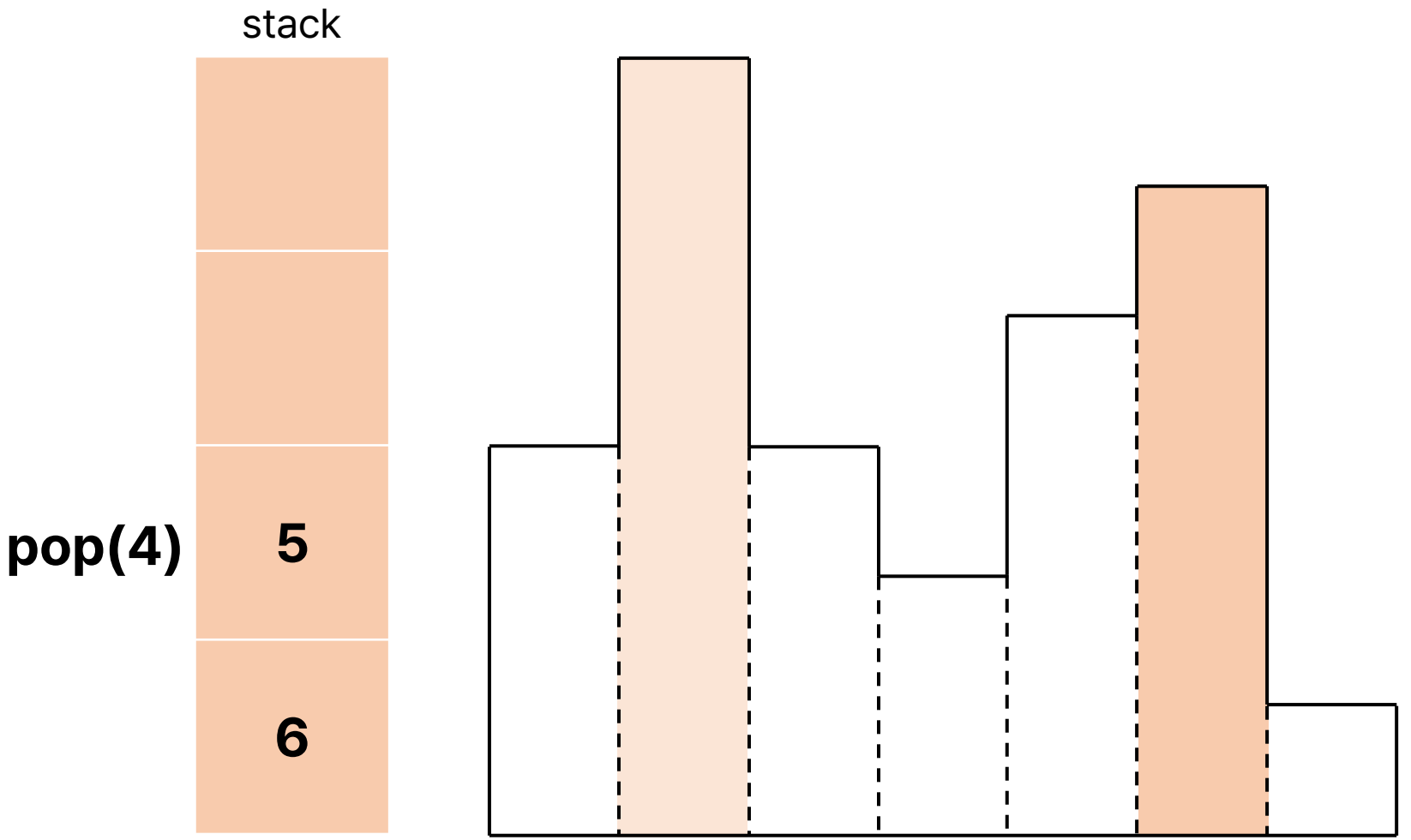
오큰수 BOJ 17298

- 더 큰 예제

  7

  3 6 3 2 4 **5** 1

  6 4 4 5



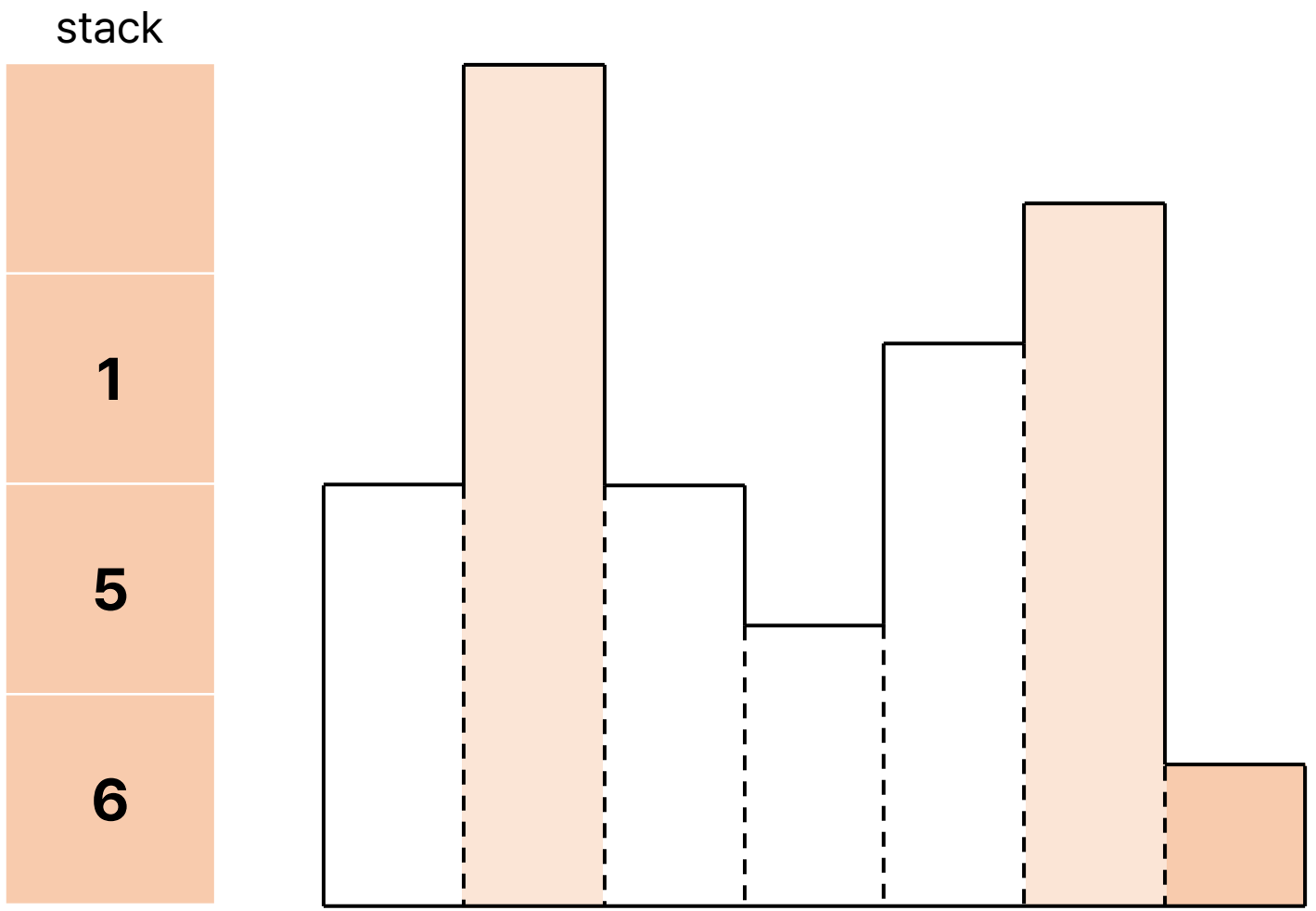
오큰수 BOJ 17298

- 더 큰 예제

  7

  3 6 3 2 4 5 1

  6 4 4 5



오큰수 BOJ 17298

- 더 큰 예제

  7

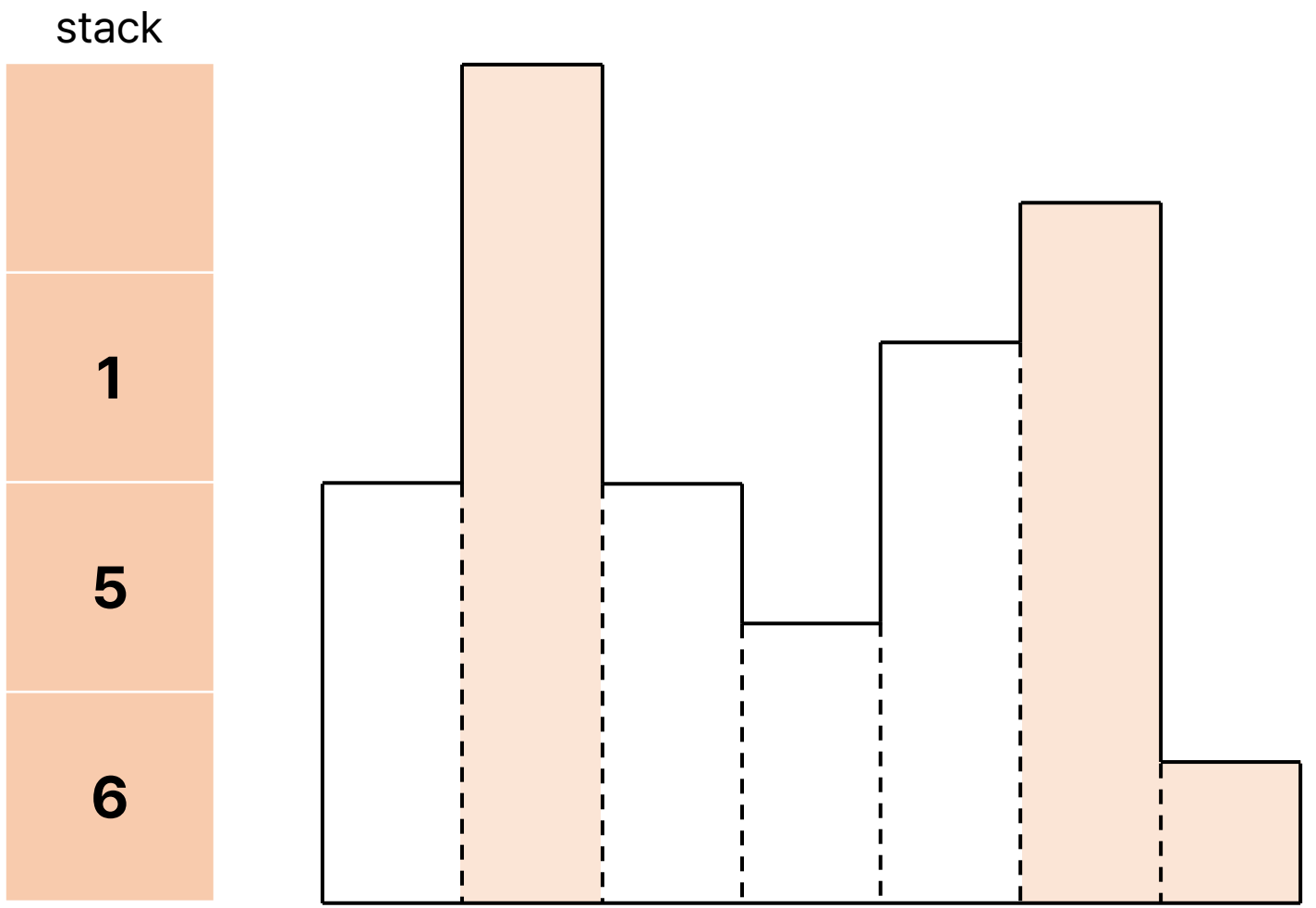
  3 6 3 2 4 5 1

  6 4 4 5

- 모든 수를 탐색한 이후

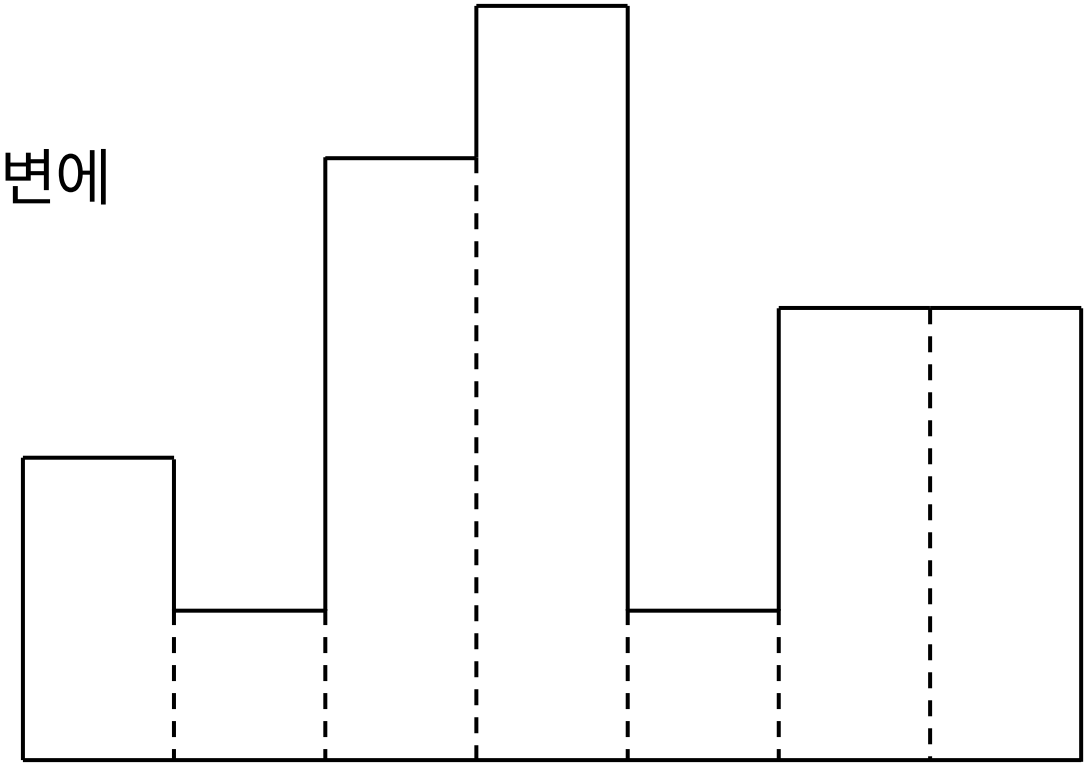
  스택에 남아 있는 수들은

  오큰수가 존재하지 않는 수



히스토그램 BOJ 1725

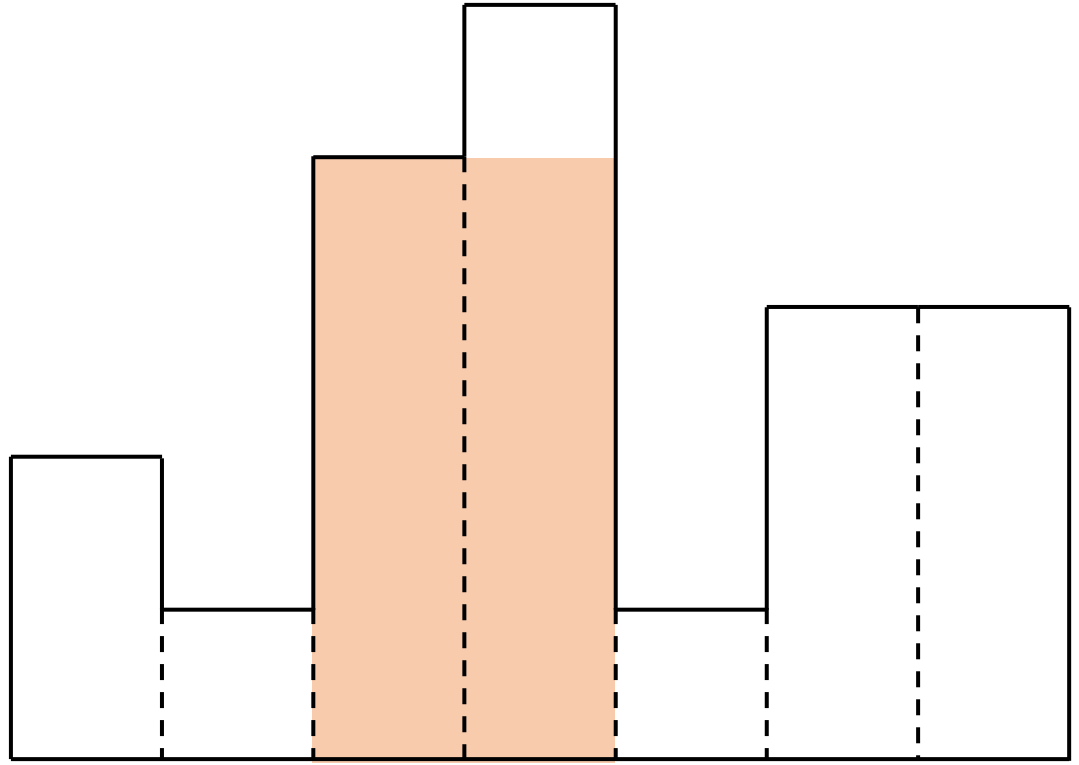
- 각 칸의 간격은 일정할 때, 히스토그램의 내부에 가장 넓이가 큰 직사각형을 구하시오
- 직사각형의 밑변은 항상 히스토그램의 아랫변에 평행하게 그려져야 한다.



히스토그램 BOJ 1725

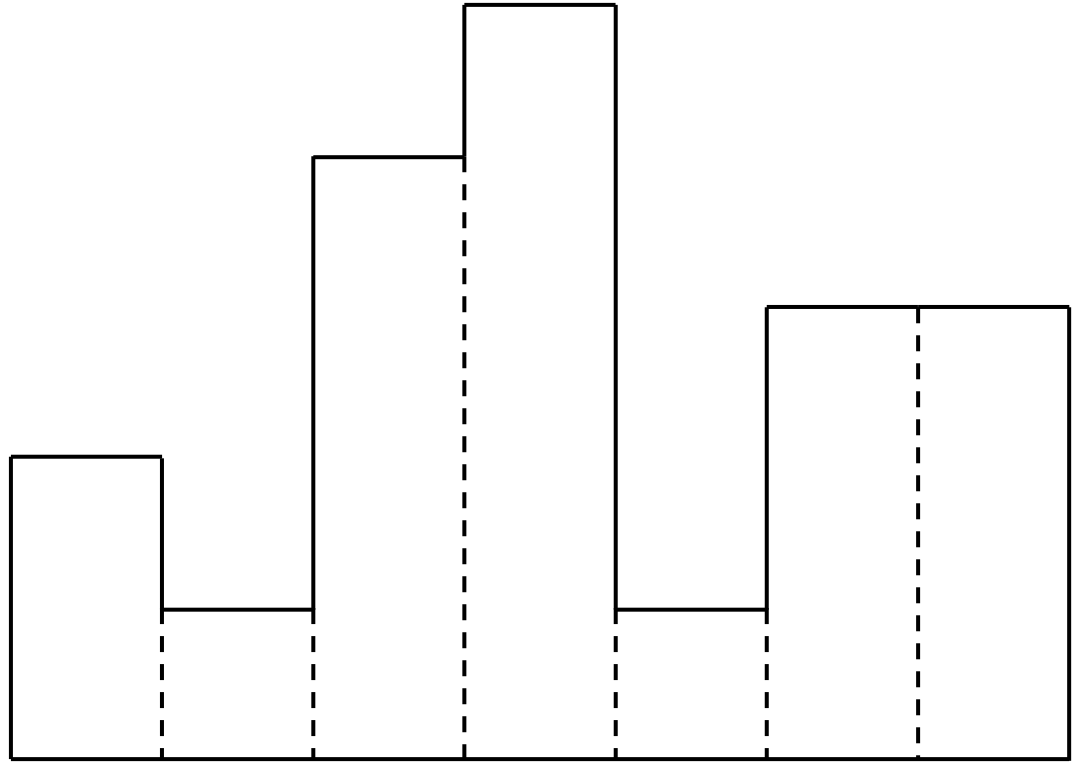
- 예제

  2 1 4 5 1 3 3



히스토그램 BOJ 1725

- 높이가 X일 때, 가능한 최대 너비를 구하자
- 너비가 X일 때, 가능한 최대 높이를 구하자



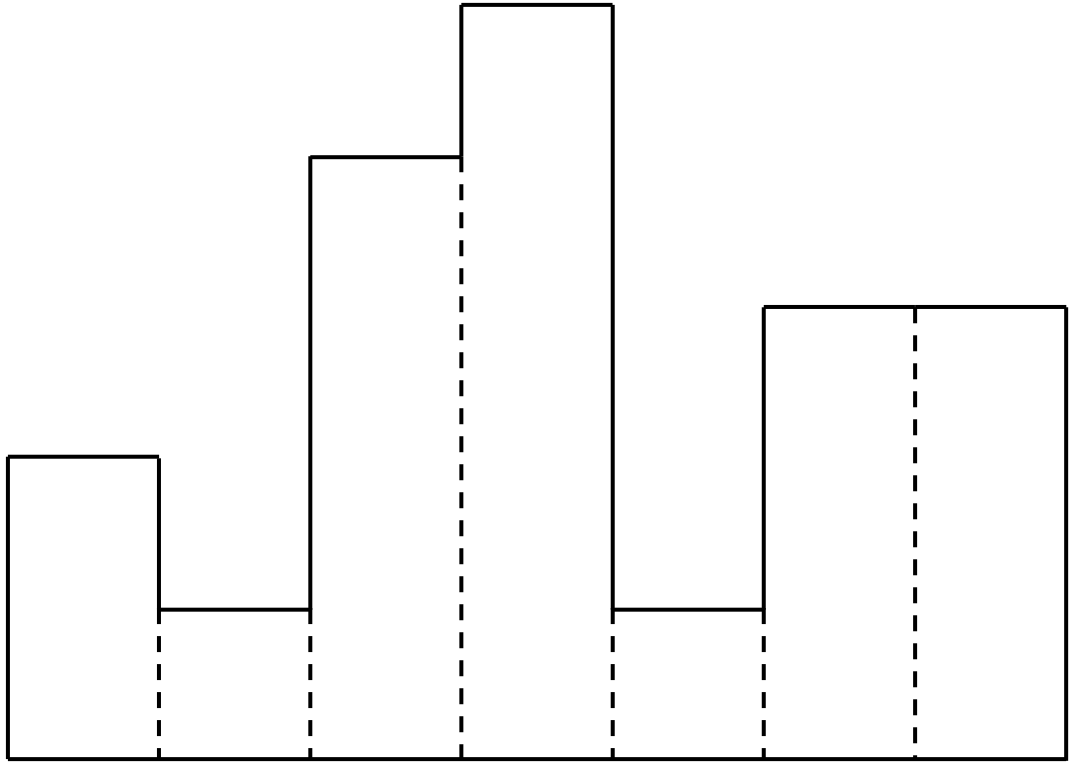
히스토그램 BOJ 1725

- 높이가 X일 때, 가능한 최대 너비를 구하자

  -> 스택

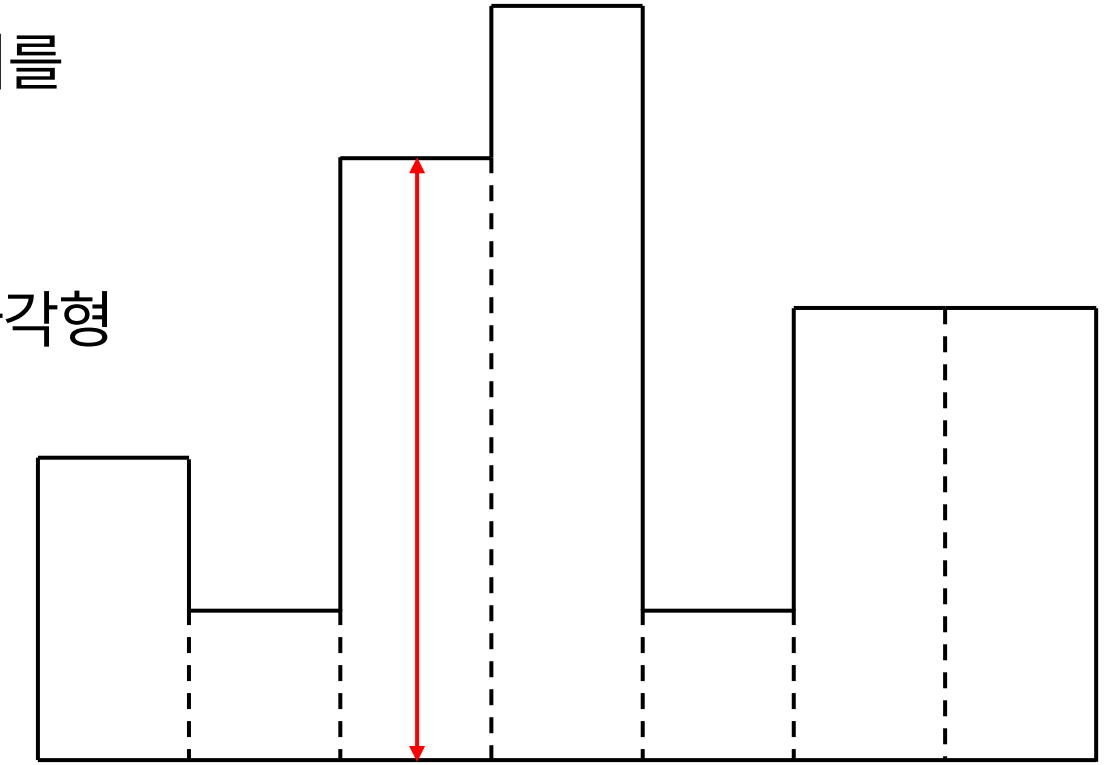
- 너비가 X일 때, 가능한 최대 높이를 구하자

  세그먼트 트리 + 분할 정복



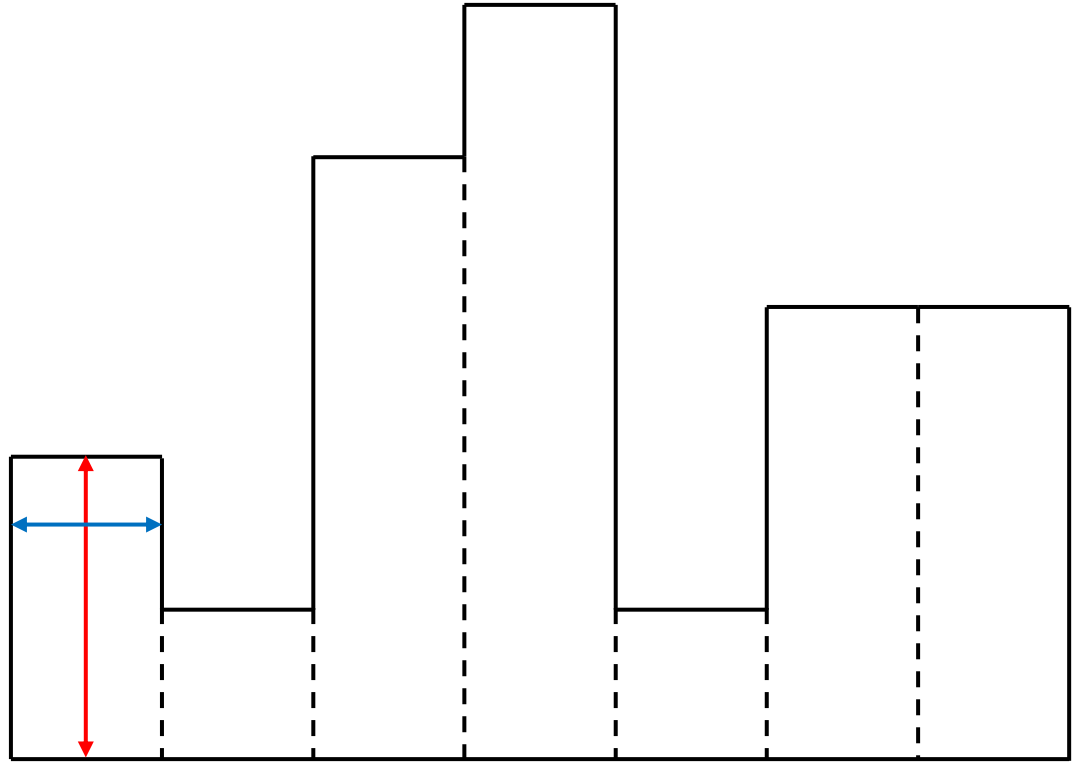
히스토그램 BOJ 1725

- 여러 개의 높이가 존재
- 각 높이를 사용했을 때 가장 긴 밑변의 길이를 구해야 한다
- i번째 높이(h_i)를 이용해 만들 수 있는 직사각형 밑변의 왼쪽 끝을 l_i, 오른쪽 끝을 r_i라 하자



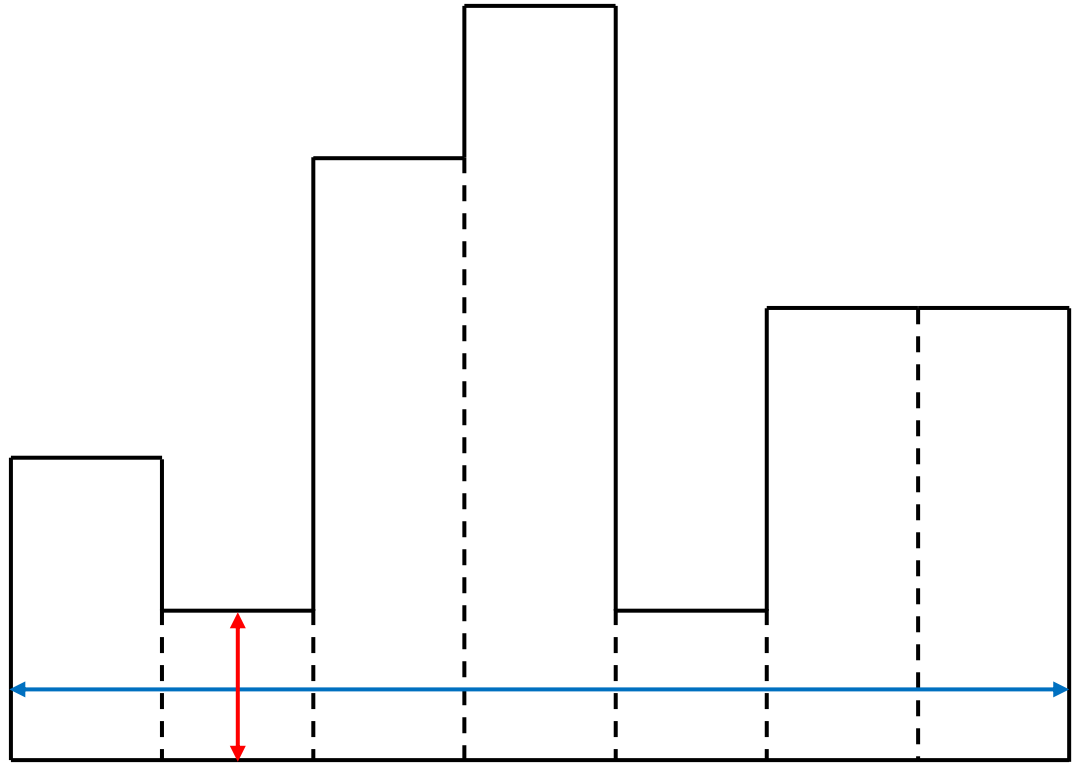
히스토그램 BOJ 1725

- $h_1 = 2, l_1 = 1, r_1 = 2$



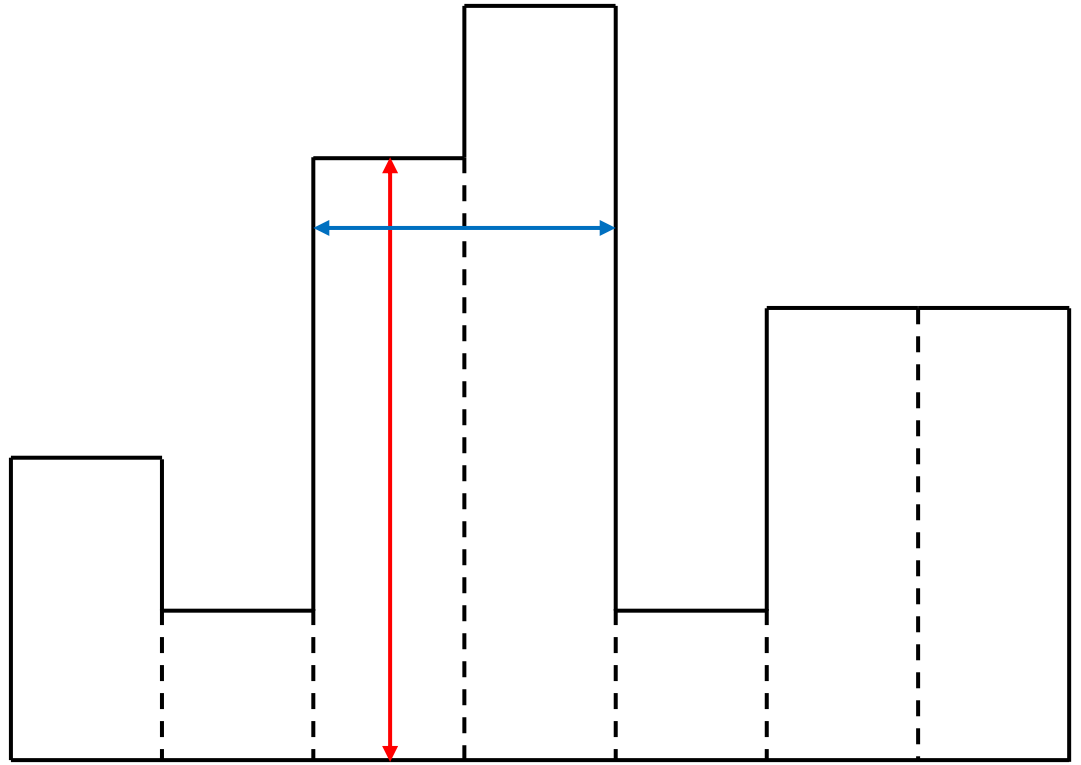
히스토그램 BOJ 1725

- $h_1 = 2, l_1 = 1, r_1 = 2$
- $h_2 = 1, l_2 = 1, r_2 = 8$



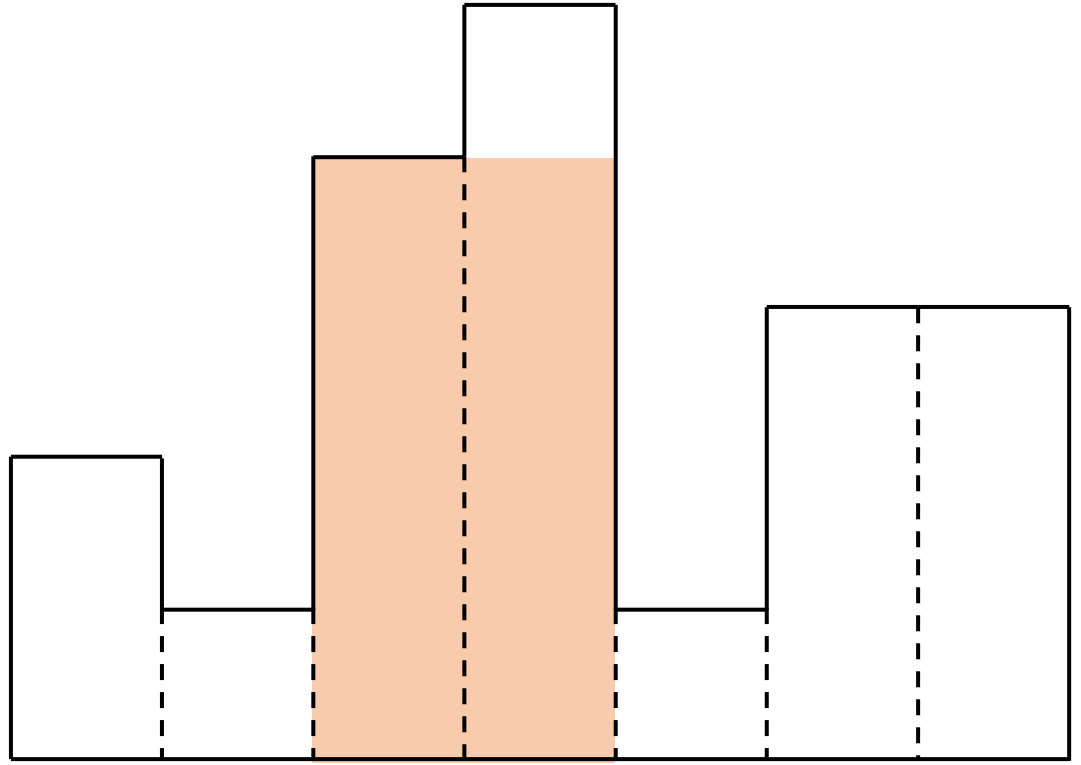
히스토그램 BOJ 1725

- $h_1 = 2, l_1 = 1, r_1 = 2$
- $h_2 = 1, l_2 = 1, r_2 = 8$
- $h_3 = 4, l_3 = 3, r_3 = 5$



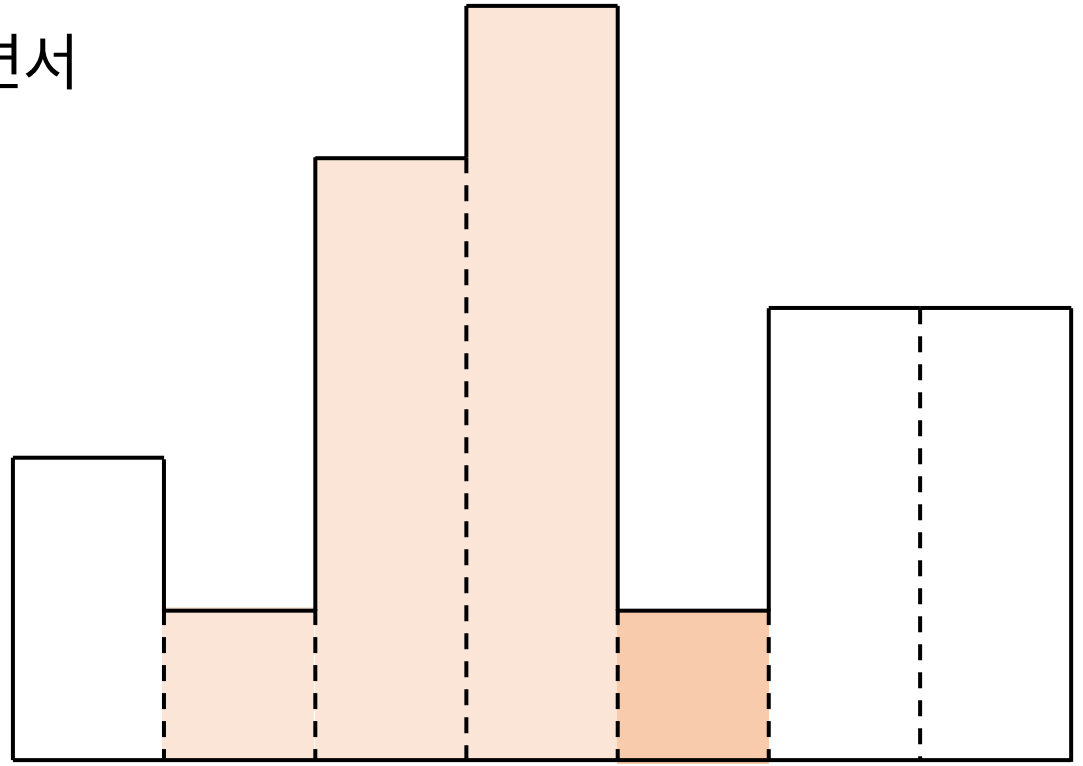
히스토그램 BOJ 1725

- $h_1 = 2, l_1 = 1, r_1 = 2 \rightarrow A_1 = 2$
- $h_2 = 1, l_2 = 1, r_2 = 8 \rightarrow A_2 = 7$
- $h_3 = 4, l_3 = 3, r_3 = 5 \rightarrow A_3 = 8$
- …



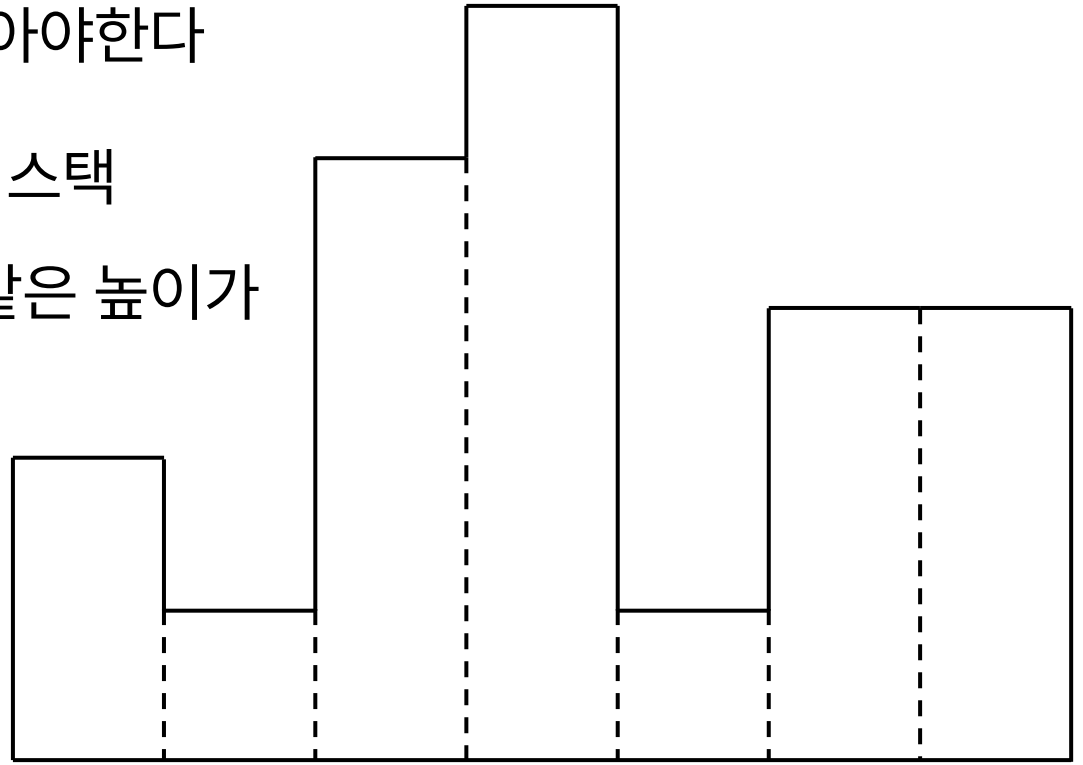
히스토그램 BOJ 1725

- r_i를 구하는 건 오큰수와 유사하다
- 내 오른쪽에 있는 높이 중에서 h_i보다 작으면서 가장 왼쪽에 있는 높이를 찾으면 된다
- 단조 증가하도록 스택을 관리한다
- r_3와 r_4는 5가 된다



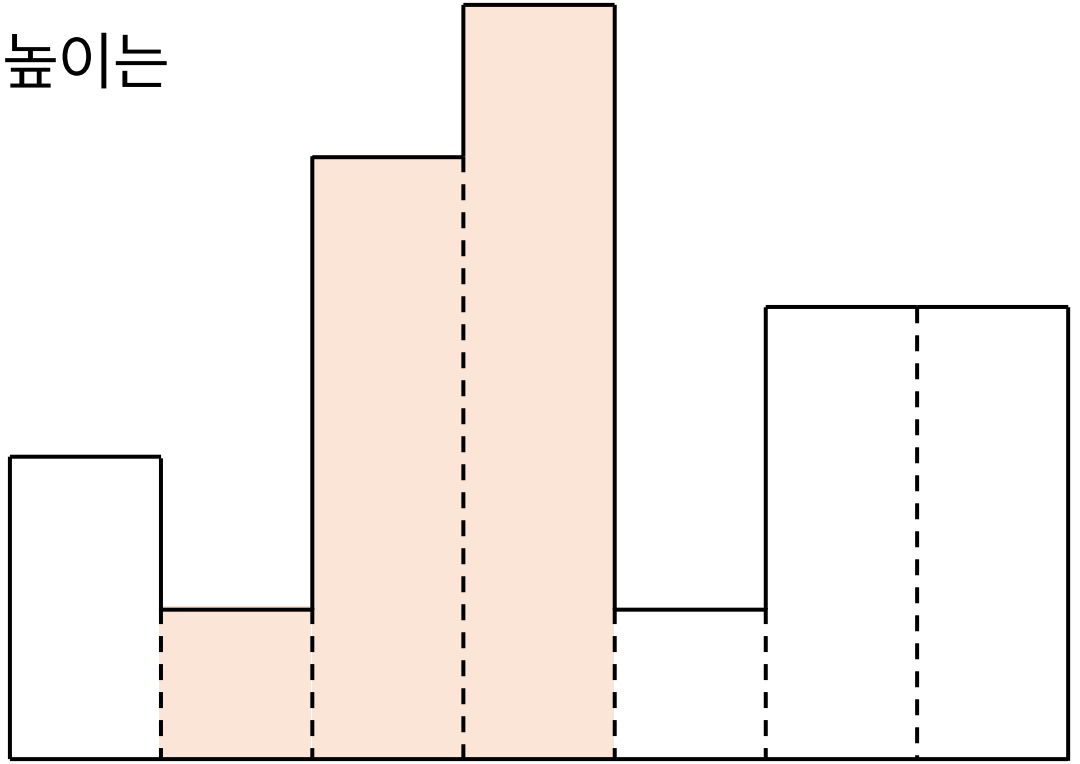
히스토그램 BOJ 1725

- l_i값을 찾을 때 현재 높이보다 왼쪽에 있는 높이 중 낮은 높이 중 가장 오른쪽에 있는 높이를 찾아야한다
- 스택에 단조 증가하도록 값을 저장했으므로 스택 현재 높이 다음에는 현재 높이보다 작거나 같은 높이가 들어있다



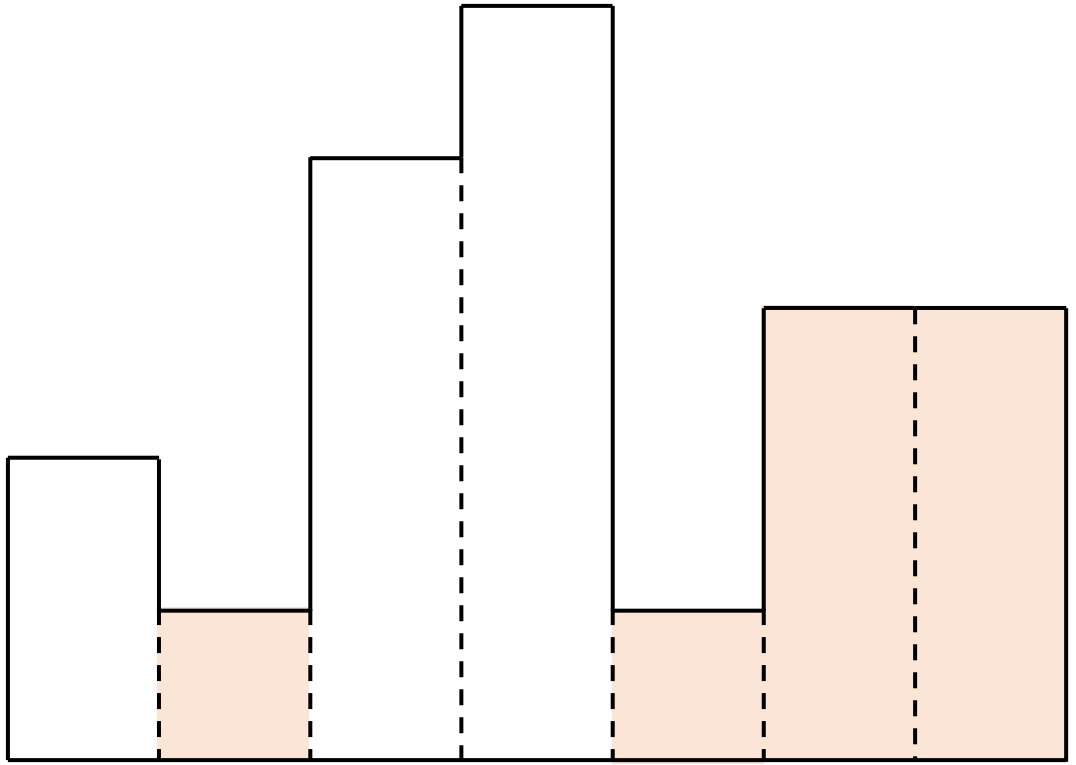
히스토그램 BOJ 1725

- 예시 1
- 5번째 높이가 들어가기 전까지 2, 3, 4번째 높이는 단조 증가한다



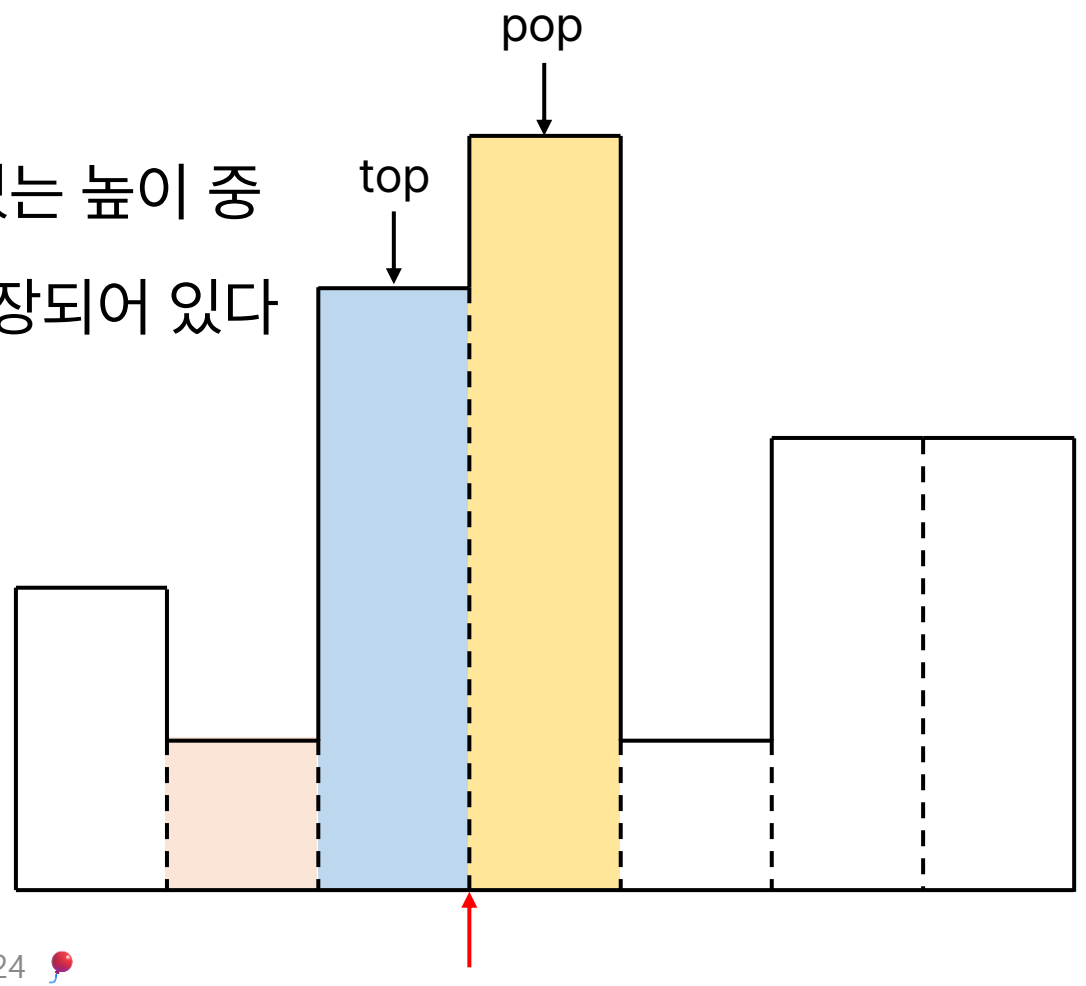
히스토그램 BOJ 1725

- 예시 2
- 5번째 높이가 들어갈 때는 3, 4번째 높이는 스택에서 제거된다
- 이후 5, 6, 7번째 높이는 단조 증가한다



히스토그램 BOJ 1725

- 높이가 같은 경우를 고려하지 않는다면
- 스택에서 pop되는 높이 다음에는 왼쪽에 있는 높이 중 낮은 높이 중 가장 오른쪽에 있는 높이가 저장되어 있다
- 즉, l_i값을 알 수 있다

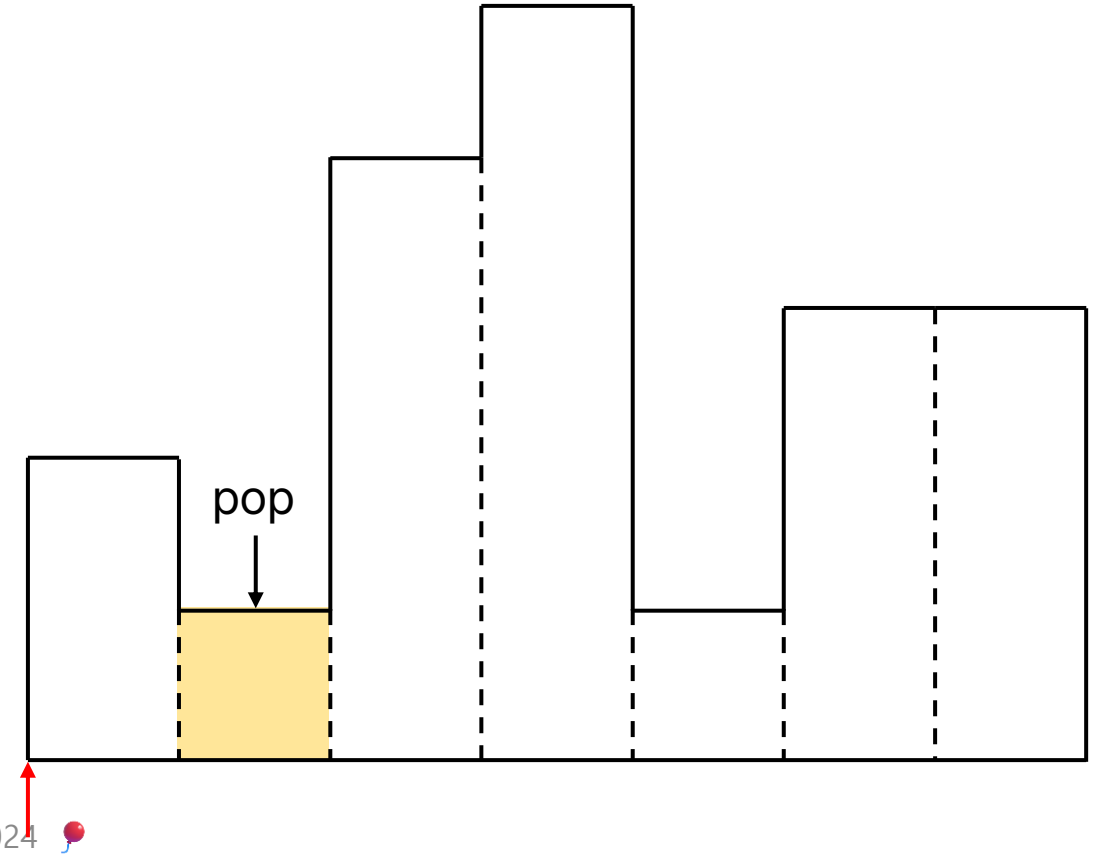


히스토그램 BOJ 1725

- 스택이 비어있다면 왼쪽에 있는 수 중에

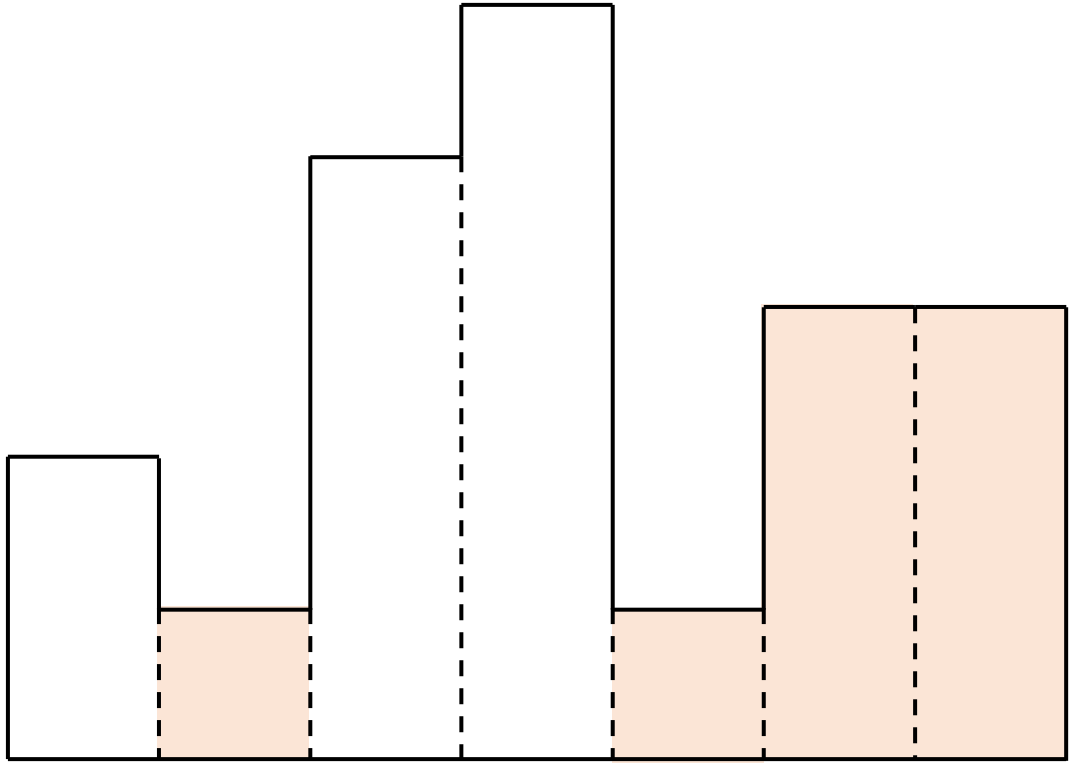
  현재 높이보다 작은 수가 존재하지 않는다

- 가장 왼쪽 경계가 l_i가 된다



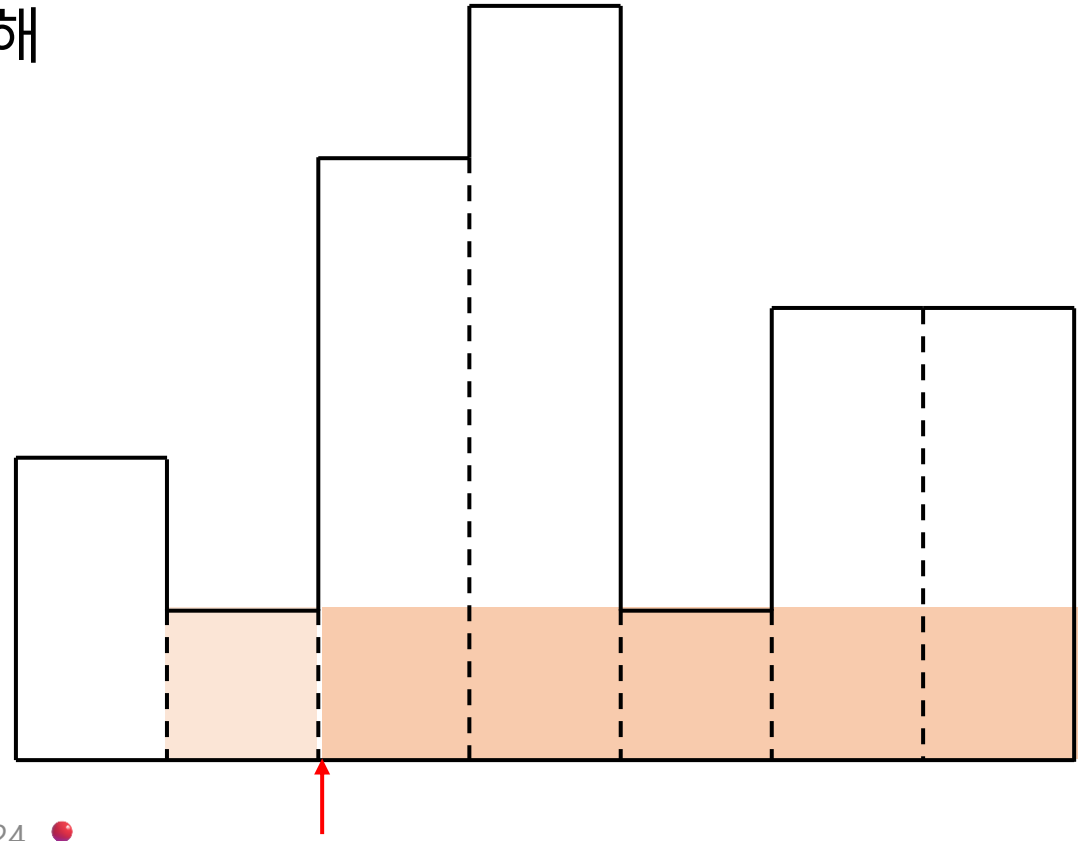
히스토그램 BOJ 1725

- 높이가 같은 경우를 고려한다면
- 예외처리를 해줄 필요가 없다
- 높이가 1인 경우를 살펴보자



히스토그램 BOJ 1725

- 예외처리를 하지 않은 경우와 동일하게 스택에 남아있는 수 중에 제일 위에 있는 값을 이용해 l_i를 구해보자
- 오른쪽에 있는 높이 1인 히스토그램인 경우 다음과 같이 직사각형이 만들어진다

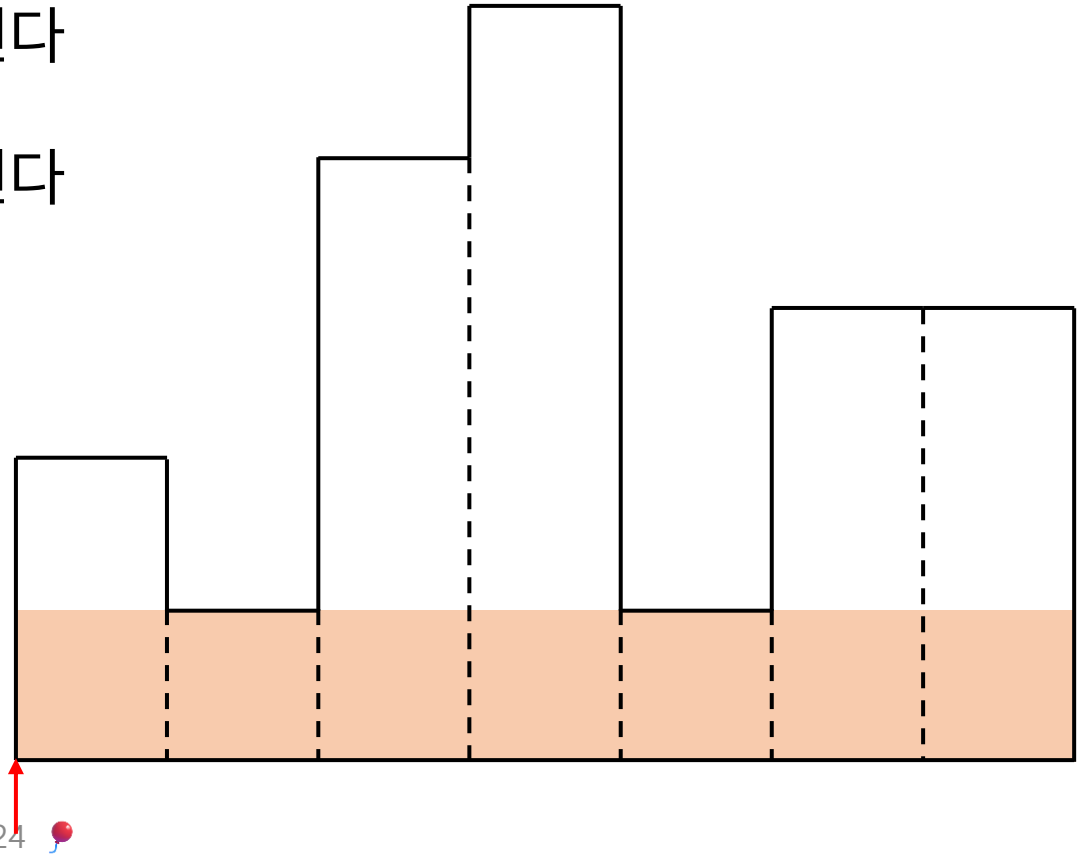


히스토그램 BOJ 1725

- 그 다음에 있는 높이가 동일한 히스토그램이

  우측에 있는 직사각형을 완전히 포함하게 된다

- 따라서 추가적인 예외처리를 하지 않아도 된다



히스토그램 BOJ 1725

- 오큰수와 유사하게 스택을 단조 증가하게 관리하면서 r_i값을 구한다
- 스택에서 h_i가 pop 될 때 r_i을 알 수 있고 동시에 l_i값을 구할 수 있다
- 따라서 pop될 때 h_i가 높이일 때 밑변의 길이를 얻을 수 있고 넓이를 계산할 수 있다
- 만일 모든 수를 살펴보고 아직 스택에 수가 남아있다면 r_i는 우측 경계가 된다

Reference

- https://github.com/justiceHui/SSU-SCCC-Study
- https://github.com/justiceHui/Sunrin-SHARC