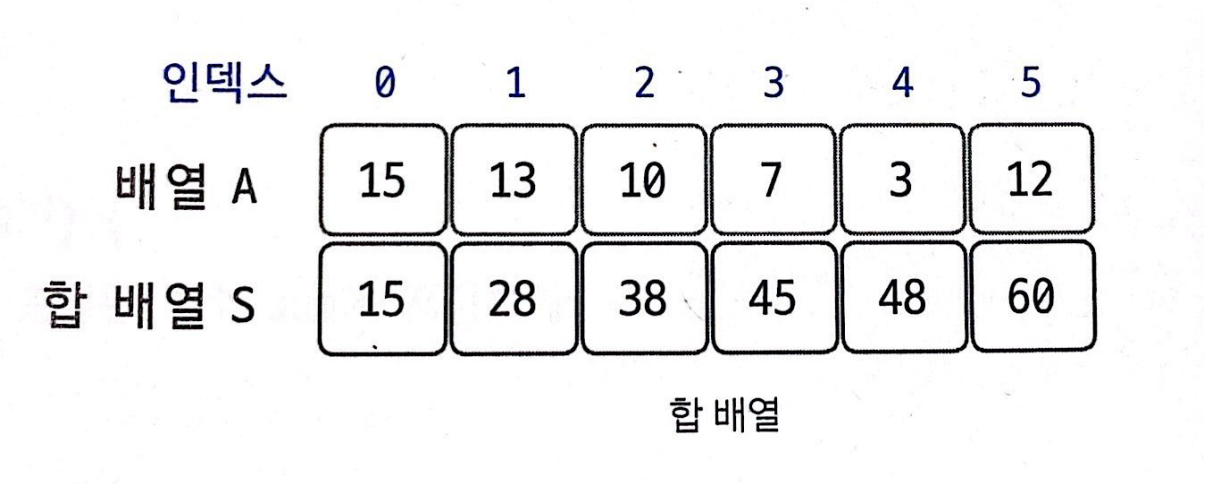
2기 / 2주차_구간 합

🌟 구간 합

합 배열을 이용해서 시간 복잡도를 O(N) → O(1)로 줄이기 위해 사용하는 알고리즘이다.

구간 합 문제란 연속적으로 나열된 N개의 수가 있을 때, 특정 구간의 모든 수를 합한 값을 구하는 문제를 말한다.

구간 합 문제를 풀 때는 접두사 합(Prefix Sum)을 사용하는 것이 유리하다.
💡 접두사 합이란 리스트의 맨 앞부터 특정 위치까지의 합을 구해 놓은 것을 의미한다.



A[i]부터 A[j]까지의 배열 합을 합 배열 없이 구할 경우 시간복잡도는 O(N)이다.
하지만 합 배열을 사용하면 O(1)에 답을 구할 수 있다.

- 합 배열 S를 만드는 공식

```
S[i] = S[i - 1] + A[i]
```

- 구간 합을 구하는 공식

```
S[j] - S[i - 1]
```

1. 백준 20438_출석체크

💡 접근법

출석 안 한 학생 수를 누적시키면 되겠다고 생각했다.

🔑 풀이

static 필드가 너무 많아서 수정하고 싶은데 한두개만 인스턴스 변수로 사용하자니 못 참겠다 .. ㅎㅎ 그렇다고 전부 뜯어 고치기엔 귀 찮 ….

먼저 졸고 있는 학생들의 출석 번호를 리스트에 담는다.

출석 코드를 받을 학생 번호를 입력 받은 후 졸고 있는 학생에 포함되지 않는다면 배수에 해당하는 출석번호도 확인한다.

초기 작업이 끝났으면 누적합을 구한다. 이전 인덱스의 값 + 현재 값인데 이 때 현재 값이라는 건 출석 안 한 학생에 해당할 경우 1을 더해준다. (출석 안 한 학생의 누적합을 구하는 것이기 때문)

구간을 입력 받고, 구간합을 출력하면 끝이다.

💻 코드

412 ms, 31112 KB

```
import java.io.*;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;

public class BOJ_20438 {
    private static final int MIN_NUMBER = 3; // 출석번호가 3번부터 시작
    private static final int NOT_ATTEND = 1; // 출석 안 함
    private static final int ATTEND = 0; // 출석 함

    private static int totalStudents; // 총 학생 수
    private static int countSleepStudents; // 조는 학생 수
    private static int codeReceiveStudents; // 출석코드를 받을 학생 수
    private static int intervalCount; // 구간 수

    private static boolean[] isReceiveCode; // 코드를 받는 학생만 true
    private static int[] totalNotAttend; // 누적합 배열
    private static List<Integer> sleepStudentsNumber; // 조는 학생 번호

    public static void main(String[] args) throws IOException {
        run();
    }

    private static void run() throws IOException {
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer st = new StringTokenizer(br.readLine());

        totalStudents = Integer.parseInt(st.nextToken());
        countSleepStudents = Integer.parseInt(st.nextToken());
        codeReceiveStudents = Integer.parseInt(st.nextToken());
```

```
        intervalCount = Integer.parseInt(st.nextToken());

        isReceiveCode = new boolean[totalStudents + MIN_NUMBER];
        totalNotAttend = new int[totalStudents + MIN_NUMBER];
        sleepStudentsNumber = new ArrayList<>();

        // 조는 학생 번호를 리스트에 담아준다.
        // 출석 코드를 받는 학생 중에 조는 학생이 있는지 판단하기 위한 용도
        st = new StringTokenizer(br.readLine());
        while (st.hasMoreTokens()) {
            int number = Integer.parseInt(st.nextToken());
            sleepStudentsNumber.add(number);
        }

        // 만약 조는 학생에 포함되지 않는다면 출석코드를 받는다고 체크하고 sendCode() 호출한다.
        st = new StringTokenizer(br.readLine());
        while (st.hasMoreTokens()) {
            int number = Integer.parseInt(st.nextToken());

            if (sleepStudentsNumber.contains(number)) {
                continue;
            } else {
                isReceiveCode[number] = true;
                sendCode(number);
            }
        }

        // 누적합을 구한다.
        // 이전 값 + 현재 학생의 출석 여부에 따라 0 혹은 1을 더해준다.
        for (int i = MIN_NUMBER; i < totalStudents + MIN_NUMBER; i++) {
            int isAttend = ATTEND;
            if (!isReceiveCode[i]) isAttend = NOT_ATTEND;
            totalNotAttend[i] = totalNotAttend[i - 1] + isAttend;
        }

        // 구간별로 출석 안한 학생 수 출력
        for (int i = 0; i < intervalCount; i++) {
            st = new StringTokenizer(br.readLine());
            int start = Integer.parseInt(st.nextToken());
            int end = Integer.parseInt(st.nextToken());

            bw.write(totalNotAttend[end] - totalNotAttend[start - 1] + "\n");
        }

        bw.flush();
        bw.close();
    }

    // 최초의 출석코드 받은 학생이 배수 번호 학생들에게 전달하는 메서드
    private static void sendCode(int number) {
        int i = 2;
        while (number * i < totalStudents + MIN_NUMBER) {
            int next = number * i++;
            if (!sleepStudentsNumber.contains(next)) {
                isReceiveCode[next] = true;
            }
        }
    }
}
```

2. 백준 21318_피아노 체조

💡 접근법

출석체크 문제랑 유사하다고 생각해서 어렵지 않게 풀이가 떠올랐다.

🔑 풀이

예제를 누적합 배열로 바꿔서 구간들을 전부 확인해봤더니 내 생각대로 결과가 나왔다.

핵심은 구간의 마지막 악보는 아무리 다음 악보보다 난이도가 높아도 결과에 포함되면 안된다는 것이다.

난이도 : [1 2 3 3 4 1 10 8 1] → 누적합 : [0 0 0 0 1 1 2 3 3] 인데,

구하려는 구간이 4~7일 경우를 보자.
7번 악보는 8번 악보보다 난이도가 높기 때문에 실수에 포함되지만, 구하려는 구간에선 8번 악보를 신경 쓰지 않기 때문에 포함하면 안된다.
따라서 애초에 4~6 구간의 구간합을 출력해야 한다.

💻 코드

672 ms, 53988 KB

```
import java.io.*;
import java.util.StringTokenizer;

public class BOJ_21318 {
    private static int countSheet; // 악보 수
    private static int countInterval; // 질문 구간 수
    private static int[] difficulty; // 난이도 배열
    private static int[] cumulativeMistake; // 누적합 배열
    private static final BufferedReader br = new BufferedReader(new InputStreamReader(System.in));


    public static void main(String[] args) throws IOException {
        input();
        parseToCumulative();
        output();
    }

    private static void output() throws IOException {
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

        for (int i = 0; i < countInterval; i++) {
            StringTokenizer st = new StringTokenizer(br.readLine());
            int start = Integer.parseInt(st.nextToken());
            int end = Integer.parseInt(st.nextToken()) - 1;

            int result = cumulativeMistake[end] - cumulativeMistake[start - 1];
            bw.write(result + "\n");
        }

        bw.flush();
        bw.close();
    }
```

```
    // 난이도 배열을 누적 합 배열로 바꾼다.
    private static void parseToCumulative() {
        cumulativeMistake = new int[countSheet + 1];
        for (int i = 1; i < countSheet; i++) {
            int mistake = 0;
            if (difficulty[i] > difficulty[i + 1]) mistake = 1;
            cumulativeMistake[i] = cumulativeMistake[i - 1] + mistake;
        }
    }

    private static void input() throws IOException {
        countSheet = Integer.parseInt(br.readLine());

        difficulty = new int[countSheet + 1];
        StringTokenizer st = new StringTokenizer(br.readLine());
        for (int i = 0; i < countSheet; i++) {
            difficulty[i + 1] = Integer.parseInt(st.nextToken());
        }

        countInterval = Integer.parseInt(br.readLine());
    }
}
```

3. 🎵니가 싫어 싫어 너무 싫어 싫어 오지 마 내게 찝쩍대지마🎵 - 1

💡 접근법

이번 주차가 구간합 알고리즘이라 열심히 생각했지만 우선순위 큐 풀이밖에 떠오르지 않았다 ,,

🔑 풀이

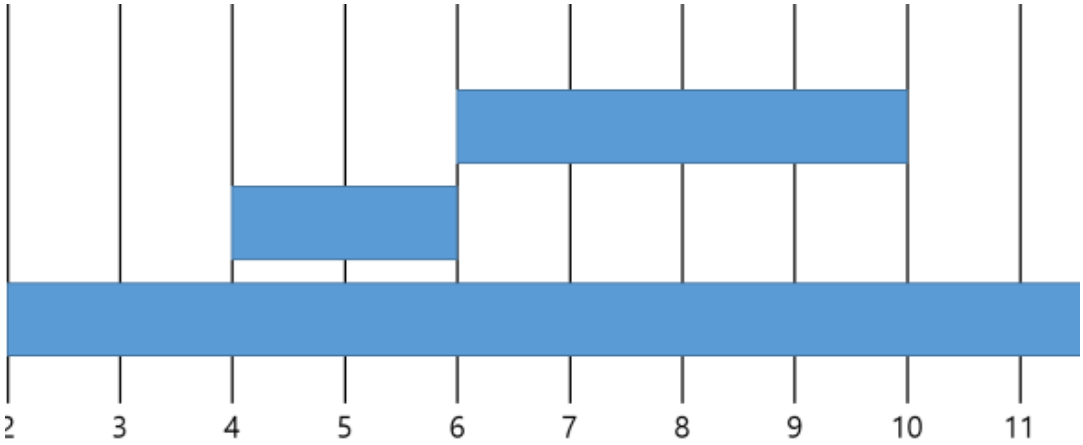
모기의 입장, 퇴장 시간은 입장 시간이 빠를수록 우선순위로/ 방에 들어온 모기는 퇴장 시간이 빠를수록 우선순위로 한다.

1. 방에 있는 모기들 중 현재 모기의 입장 시간보다 퇴장 시간이 빠르면 방에서 내보낸다.

```
Mosquito current = mosquito.poll();
    while (!room.isEmpty() && room.peek().exitTime <= current.admissionTime) {
        room.poll();
    }
```

2. 현재 모기를 방에 들여보낸다.
3. 방에 있는 모기의 수를 확인해서 최댓값을 갱신한다.

간단하게 생각했지만 예제처럼 이어지는 구간 처리가 생각보다 어려웠다.



일단 방에 있는 모기 수가 최댓값보다 크다면 구간의 시작시간, 종료시간, 모기 최댓값 전부 갱신한 후 이어지는 구간인지 다시 확인해서 종료시간만 다시 갱신했다.

💻 코드

1228 ms, 263976 KB

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.PriorityQueue;
import java.util.StringTokenizer;

public class BOJ_20440 {
    private static int totalMosquito;
    private static long maxMosquito = 0;
    private static long maxStartTime, maxEndTime;
    // 모기 (입장시간 기준 정렬)
    private static PriorityQueue<Mosquito> mosquito = new PriorityQueue<>((r1, r2) -> {
        if (r1.admissionTime > r2.admissionTime) return 1;
        else return -1;
    });
    // 방 (퇴장시간 기준 정렬)
    private static PriorityQueue<Mosquito> room = new PriorityQueue<>((r1, r2) -> {
        if (r1.exitTime > r2.exitTime) return 1;
        else return -1;
    });

    private static class Mosquito {
        long admissionTime;
        long exitTime;

        public Mosquito(long admissionTime, long exitTime) {
            this.admissionTime = admissionTime;
            this.exitTime = exitTime;
        }
    }

    public static void main(String[] args) throws IOException {
        input();
        solve();
        output();
```

```
    }

    private static void output() {
        System.out.println(maxMosquito);
        System.out.println(maxStartTime + " " + maxEndTime);
    }

    private static void solve() {
        while (!mosquito.isEmpty()) {
            Mosquito current = mosquito.poll();
            while (!room.isEmpty() && room.peek().exitTime <= current.admissionTime) {
                room.poll();
            }
            room.offer(current);

            // 방에 있는 모기 수가 최대값보다 크면 max값들 갱신하기
            if (maxMosquito < room.size()) {
                maxMosquito = room.size();
                maxStartTime = current.admissionTime;
                maxEndTime = room.peek().exitTime;
            }

            // 최대 모기 수가 같으면서 이어지는 구간 확인해서 maxEndTime 갱신하기
            if (maxMosquito == room.size() && maxEndTime == current.admissionTime) {
                maxEndTime = room.peek().exitTime;
            }
        }
    }

    private static void input() throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        totalMosquito = Integer.parseInt(br.readLine());

        for (int i = 0; i < totalMosquito; i++) {
            StringTokenizer st = new StringTokenizer(br.readLine());
            long admissionTime = Long.parseLong(st.nextToken());
            long exitTime = Long.parseLong(st.nextToken());
            mosquito.offer(new Mosquito(admissionTime, exitTime));
        }
    }
}
```