🎶

세그먼트 트리 - Segment Tree

왜 세그먼트 트리를 사용하는가?

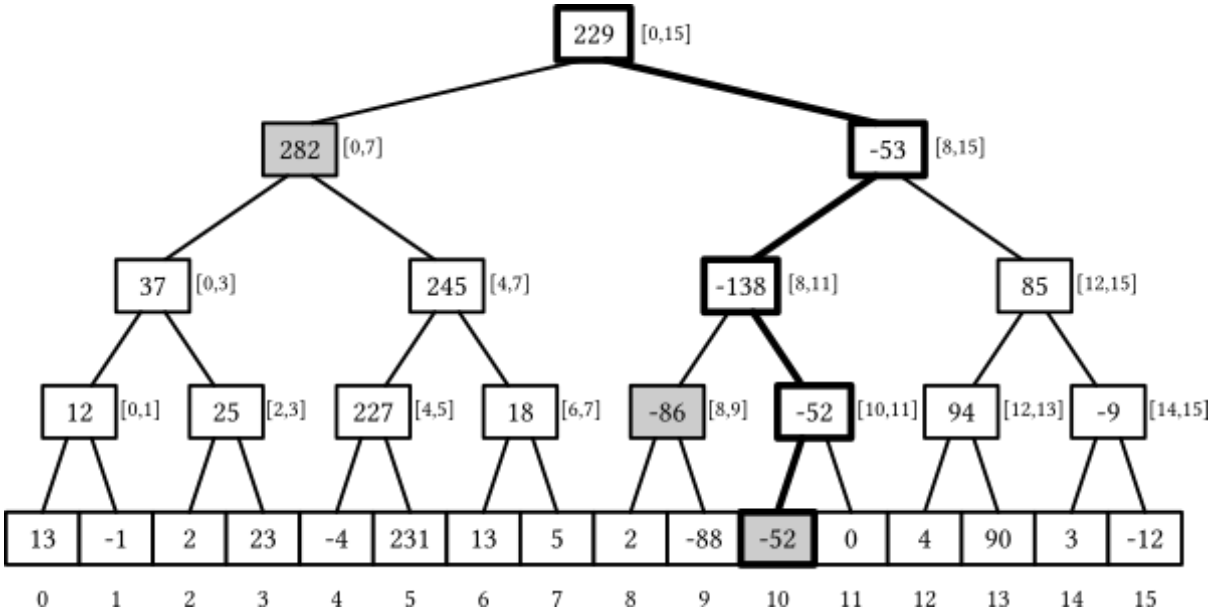
> 💡 Segment : [명사] 부분 (from. 네이버 영어사전)

`세그먼트 트리`는 배열, 리스트 등이 있을때 **특정 구간에서 합, 곱, 차이 등이 어떻게 되는지를 저장하는 트리**입니다. 예를 들어 배열 { 1, 2, 3, 4, 5 } 가 있다면 2번째~4번째 원소의 합은 어떤가를 구해주는 거라 생각하면 됩니다.

위의 경우에서 단순히 한번만 구간 연산을 한다면 따로 이런 자료구조를 만들 필요가 없습니다. 그냥 $O(n)$ 시간 복잡도로 탐색을 하면 되니깐요. 그런데 구간 합을 자주 구하게 될 수록 선형 시간 복잡도가 계속 쌓이게 되고 결국 엄청난 시간을 잡아먹게 될 것입니다.

이 경우 `세그먼트 트리`를 사용하게 된다면 $O(logN)$의 시간으로 줄여서 연산을 할 수 있게 됩니다!

트리로 보는 세그먼트 트리



이미지 출처 : https://en.algorithmica.org/hpc/data-structures/segment-trees/

위 그림은 세그먼트 트리 중에서 `구간 합`을 구하는 세그먼트 트리를 나타낸 것입니다. 총 원소가 16개 있으며 트리를 보면 [from, to]로 구간 합을 나타내는걸 볼 수 있습니다. 그리고 여기서 `리프 노드`는 기존 배열의 원소임을 확인할 수 있습니다.

이제 이런 식의 세그먼트 트리를 코드로 표현하게 되는 것입니다.

세그먼트 트리의 크기

세그먼트 트리를 만들기 전 원소가 n개 있을때 트리의 크기를 구할 수 있어야 합니다.

위 그림을 예시로 하면 n=16이 되고 트리의 높이는 4가 됩니다. $log_2 16$로 계산할 수 있는 것이죠. 여기서 만들어지는 세그먼트 트리는 `이진 트리`로 자식을 항상 2개까지만 가질 수 있는 트리입니다. 때문에 로그를 통해 간단하게 구할 수 있는데 여기서 주의할 점이라면 **로그를 취한 뒤 올림을 해 줘야 높이가 나온다**는 것입니다. 그 이유는 다들 잘 아실거로 생각됩니다.

이제 높이를 구했으니 세그먼트 트리에는 총 몇개의 원소가 들어가는지 크기를 구해야 합니다. 정답부터 말씀드리자면 `1 << (위에서 구한 트리 높이 + 1)` 입니다. 이건 $2^{높이} * 2 = 2^{높이+1}$을 C++로 나타낸 것입니다.

위 그림에서 n=16일때 총 15 + 16 = 31개의 노드들이 만들어 진걸 확인할 수 있습니다. 이는 완전 이진 트리를 만들때 **높이에서 나올 수 있는 최대 원소 + 1**를 표시한 것입니다.

그런데 n이 2의 거듭제곱이 아니면 꽤 많은 배열이 비어있게 됩니다. 만일 n=5 라고 하면 높이는 3(2.... 에서 올림)이 나오게 되고 위 식대로 크기를 구하면 16개가 나오게 됩니다. 그런데 실제로 구간합을 위한 세그먼트 트리를 뽑아보면 9개만 사용하게 되며 결과적으로 7개가 낭비됩니다.

> 💡 n=5 일때 트리의 구성
> - 기존 원소 5개
> - 1~2 합
> - 1~3, 4~5 합
> - 전체 합

하지만 이렇게 정확한 크기를 구하는데는 시간이 많이 들게 되고 결과적으로 빠르게 해결하기 위해서 위 식을 사용하게 된 것입니다. 위 식이 2의 거듭제곱에서 딱 맞는 크기거든요.

코드로 보는 세그먼트 트리

세그먼트 트리에서는 기본적으로 총 3가지 연산이 있어야 합니다.

- 트리 만들기
- 구간 연산
- 트리 업데이트

알고리즘 문제를 풀이하다 보시면 특정 인덱스의 원소가 변경되는 경우가 있는데 세그먼트 트리를 활용하게 되면 이를 빠른 시간에 업데이트 해 줄 수 있게 됩니다.

트리 만들기

세그먼트 트리의 그림을 보면 맨 아래 이진 트리에서 구간합을 구해오면서 합쳐오는 방식입니다. 이는 `재귀함수`를 통해서 구현할 수 있습니다.

```
int InitTree(vector<int> &numbers, vector<int> &tree, int node, int start, int end){
    if(start == end)
```

```
        return tree[node] = numbers[start];

    int mid = (start + end) / 2;

    return tree[node] = InitTree(numbers, tree, node*2, start, mid) +
                        InitTree(numbers, tree, node*2+1, mid+1, end);
}
```

위 코드가 재귀함수로 표현한 것으로 함수는 **원래 원소 배열, 세그먼트 트리로 저장할 것, 현재 노드, 시작, 끝**을 알고 있어야 합니다.

위 함수의 동작은 간단하게 세그먼트 트리의 리프 노드(= 원래 원소)까지 내려간 뒤 올라오면서 하나씩 합치는 것입니다. 이진 트리임을 이용해 자신의 왼쪽, 오른쪽 자식의 값을 더하면서 완성하는 것이죠.

함수 호출 모습을 보면 트리의 아래를 찍은 뒤 올라오는 모습입니다.

구간 연산

이번 항목에서는 이해를 돕기 위해 `구간 합`을 구하는 것으로 코드를 만들어 보겠습니다.

```
// node : 현재 보고 있는 세그먼트 트리의 노드(인덱스)
// start, end : 탐색할 범위
// left, right : 원하는 구간의 왼쪽, 오른쪽 인덱스
// node : 현재 세그먼트 트리의 노드 번호
int GetPrefixSum(vector<int> &tree, int node, int start, int end, int left, int right){
    if(left > end || right < start)
        return 0;
    if(left <= start && end <= right)
        return tree[node];

    int mid = (start + end) / 2;
    return GetPrefixSum(tree, node*2, start, mid, left, right) +
           GetPrefixSum(tree, node*2+1, mid+1, end, left, right);
}
```

구간 연산을 할 때 아래 3가지 경우가 있는데

- 탐색중인 곳이 원하는 범위가 완전히 아닐때 (1)
- 탐색중인 곳이 원하는 범위 안에 있을 때 (2)
- 탐색중인 곳과 원하는 범위가 일부 겹칠때 (3)

위 (1) 연산의 경우 위 함수에서 return 0를 하는 것으로 표현되어 있습니다. 탐색할 필요가 없으니 그냥 0을 리턴하는 것이죠.

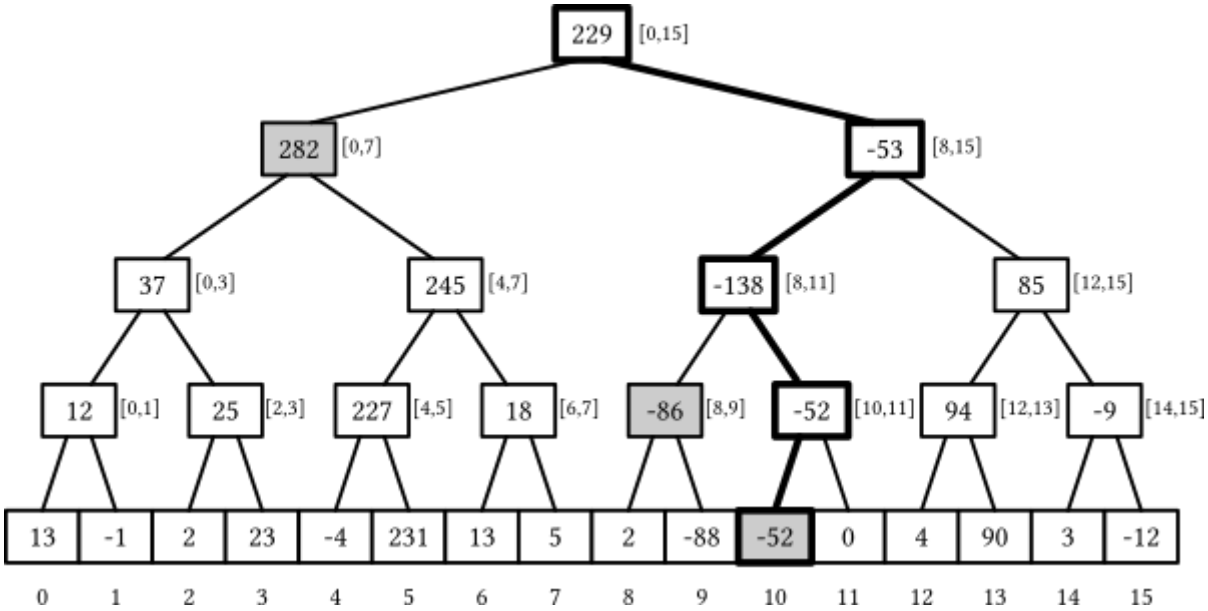
(2) 번 연산의 경우 바로 밑 `if(left <= start && end <= right)` 로 표현되어 있습니다. 이 경우 현재 탐색중인 **세그먼트 트리의 인덱스**를 반환하면 되는 것입니다.

(3) 번의 경우 (2) 가 나올 수 있도록 더 깊은 연산을 해 줘야 합니다. 때문에 왼쪽, 오른쪽으로 나누어서 재귀 함수를 호출하게 되는 것입니다.

> 💡 함수 호출을 보면 위에서부터 아래로 내려가면서 연산하는 모습입니다.

값 업데이트

간혹 기존 원소 배열에서 값을 바꾸는 괴랄한 연산들이 있습니다. `세그먼트 트리`의 경우 한 인덱스를 변경했다고 하면 그 위에 영향을 받는 인덱스도 변경해야 합니다.



만일 위 값에서 10번 인덱스인 -52를 변경했다면 이의 영향을 받는 모든 트리 노드를 변경해 줘야 하는 것입니다. 이 경우 **2가지 경우**를 앞두게 됩니다.

- 영향을 받는 경우

- 받지 않는 경우

```
// node : 현재 보고 있는 세그먼트 트리의 노드(인덱스)
// start, end : 현재 탐색중인 범위
// index : 변경된 인덱스
// difference : 기존 값과의 차이
void UpdateTree(vector<int> &tree, int node, int start, int end,
                int index, int difference){
    if(!(start <= index && index <= end))
        return;

    tree[node] += difference;

    if(start != end){
        int mid = (start + end) / 2;
        UpdateTree(tree, node*2, start, mid, index, difference);
        UpdateTree(tree, node*2+1, mid+1, end, index, difference);
    }
}
```

- `if(!(start <= index && index <= end))`
  - 영향을 받지 않는 경우로 바로 return 하게 됩니다.
- `tree[node] += difference`
  - 위 분기문에서 걸러지지 않았다면 영향을 받는 곳이므로 업데이트 해 줍니다.
  - `if(start != end)` : 이는 리프노드가 아님을 의미합니다
    - 만일 리프 노드가 아니라면 안쪽으로 들어가면서 리프 노드가 나올 때 까지 업데이트 해 줘야 합니다.
    - 이를 위해서 현재 탐색중인 범위를 좌, 우로 나누어서 탐색하게 됩니다.

함수 호출 모습을 보면 위 구간 연산처럼 위에서 아래로 내려가는 모습입니다.

추천 문제

2042번: 구간 합 구하기

첫째 줄에 수의 개수 N(1 ≤ N ≤ 1,000,000)과 M(1 ≤ M ≤ 10,000), K(1 ≤ K ≤ 10,000) 가 주어진다. M은 수의 변경이 일어나는 횟수이고, K는 구간의 합을 구하는 횟수이다. 그리고 둘째 줄부터 N+1번째

https://www.acmicpc.net/problem/2042

2357번: 최솟값과 최댓값

https://www.acmicpc.net/problem/2357

11505번: 구간 곱 구하기

첫째 줄에 수의 개수 N(1 ≤ N ≤ 1,000,000)과 M(1 ≤ M ≤ 10,000), K(1 ≤ K ≤ 10,000) 가 주어진다. M은 수의 변경이 일어나는 횟수이고, K는 구간의 곱을 구하는 횟수이다. 그리고 둘째 줄부터 N+1번째

https://www.acmicpc.net/problem/11505