CS202: 고급 자료구조 보고서

20160385 오진영

1. Purifier

이 문제를 단순한 DP 문제로 생각하게 되면, 공기청정기의 배열을 앞에서부터 순회하면서 두 값을 update 하면서 풀 수 있는데, 그 두 값은 $exclude_i$ = i 번째 공기청정기를 사용하지 않는 가장 큰 성능의 합, $include_i$ = i 번째 공기청정기를 사용하는 가장 큰 성능의 합 이다. 이때 점화식은 $exclude_i$ = max(exclude $exclude_{i-1}$, $include_{i-1}$), $include_i$ = $exclude_i$ + a_i 가 된다. 이렇게 하게 되면 O(nm)시간 안에 문제를 해결할 수 있지만, 그렇게 되면 100 점을 받을 수 없을 것이라고 생각했고 수업 시간에 했던 segment tree 를 적용하여 O(mlogn + nlogn)시간에 처리할 수 있는 방법을 생각해 내었다. 수업 시간에 푼 문제에서는 구간합과 최솟값을 tree 의 각 node 에 저장했으나, 이 문제를 풀기 위해서는 위의 점화식과 유사한 규칙으로 만들어 진 include 와 exclude 값들을 저장하면 된다. 단, tree 의 부모 node 로 올라갈 때 두 자식 node 에 있는 purifier 사용 정보를 합쳐야 하기 때문에 끝 부분의 purifier 의 포함(include)/불포함(exclude)뿐만 아니라 첫 purifier 의 포함 여부 역시 알아야만 한다. 즉, tree 의 각 node 는 그 구간이 [i, j]일 때 아래와 같은 네 값을 가져야 하는 것이다.

1. (i, j): 양 끝의 purifier 를 모두 사용하지 않음
2. [i, j): 마지막 purifier 를 사용하지 않음
3. (i, j]: 첫 purifier 를 사용하지 않음
4. [i, j]: 양 끝 purifier 를 모두 사용함

그리고 부모 node 에서 합칠 때에는 연속해서 두 purifier 를 사용할 수 없으므로 j] 인 왼쪽 자식 node 와 [i 인 오른쪽 자식 node 는 합쳐질 수 없다. 나머지 경우의 수들에 대해서 각각 성능의 합을 최대로 하는 경우를 사용하면 되는데, 이를 코드로 나타내면 아래와 같다.

```
tree[p].inin = MAX(
                    tree[2*p].inex + tree[2*p+1].inin,
                    tree[2*p].inin + tree[2*p+1].exin,
                    tree[2*p].inex + tree[2*p+1].exin
    );
    tree[p].inex = MAX(
                    tree[2*p].inex + tree[2*p+1].inex,
                    tree[2*p].inin + tree[2*p+1].exex,
                    tree[2*p].inex + tree[2*p+1].exex
    );
    tree[p].exin = MAX(
                    tree[2*p].exin + tree[2*p+1].exin,
                    tree[2*p].exex + tree[2*p+1].exin,
                    tree[2*p].exex + tree[2*p+1].inin
    );
    tree[p].exex = MAX(
                    tree[2*p].exin + tree[2*p+1].exex,
                    tree[2*p].exex + tree[2*p+1].exex,
                    tree[2*p].exex + tree[2*p+1].inex
    );
```

이 때 말단 node 의 경우에는 자기 자신을 항상 포함하는 경우만 존재할 수 있다고 두고 inin 에만 값을 할당하고 나머지는 최소값인 0 을 두게 되면 알아서 inin 만으로 부모 node 의 값이 채워지게 된다. 값을 update 할 때 마다 위와 같은 방식으로 root node 까지 update 하게 되면 root node 에는 전체 purifier 에 대해서 성능을 최대로 하기 위한 네 가지 경우의 수의 값들이 존재하게 되고, 단순하게 네 값들의 최대값을 찾아 출력하면 각 날의 최대 성능 값이 된다.

2. Errand

이 문제는 위의 문제와 유사하지만 조금 더 복잡하다. 위의 문제는 연속한 purifier 들을 사용하지 않기 위해서 양 끝의 purifier 의 사용 여부에 따라서 네 가지 경우로 분류해서 각 node 마다 값을 가지고 있다가 부모 node 로 합치면서 연속하여 purifier 를 사용하는 경우가 없도록 하였다면, 이 문제는 사이의 하나의 장소만 패스할 수 있기 때문에 그 하나의 장소가 위치하는 곳에 따라서 더 복잡한 경우가 필요하다.

나는 이 문제를 해결하기 위하여 각 node 의 segment 에 대해 s, m, e, f 라는 네 개의 Path 구조체를 할당하였다. 우선 Path 구조체는 해당하는 segment 의 범위에 있는 모든 장소들을 방문할 때, 그 시작 점과 마지막 점의 좌표와 사이의 거리를 저장한다. S path 는 segment 의 범위에 있는 장소들 중 첫 번째 장소를 패스하고 나머지 장소들을 방문하는 경우에 대한 것이고, M path 는 범위에 있는 장소들 중 처음이나 마지막 장소가 아닌 중간에 있는 장소들 중 하나를 패스하는 경우 중 최단거리를 보장하는 경로에 대한 것이고, E path 는 범위의 마지막 장소를 패스하고 나머지 장소들을 방문하는 경우에 대한 것이다. 마지막으로 F path 는 패스 없이 모든 장소를 방문하는 경우에 대한 것이다.

이때 부모 node 에서 자식 node 의 네 가지 path 를 참조해 부모 node 의 네 가지 path 를 만들 때는 각 path 에서 한 장소만 패스하고 나머지를 모두 방문하도록 주의해야 한다. 예를 들자면, 부모 node 의 S path 는 왼쪽 자식 node 의 s path 와 오른쪽 자식 node 의 f path 을 합친 것이어야만 맨 첫 장소를 패스하는 path 를 구할 수 있게 된다. 마찬가지로 E path 는 왼쪽 자식의 f path 와 오른쪽 자식의 e path 를 합치면 되고 F path 는 모든 장소를 방문하는 것이므로 두 자식의 F path 를 합치면 된다. 다만 M path 의 경우에는 조금 복잡한데, 왼쪽 자식의 마지막 장소나 오른쪽 자식의 첫 장소는 부모 node 의 기준에서 보았을 때 가운데에 위치하는 장소이므로 왼쪽 자식의 m, e path 와 오른쪽 자식의 s, m path 들의 조합 중 가장 최단거리를 택하면 된다. 코드로 나타내면 아래와 같다.

```
tree[p].s = mergePath(tree[2*p].s, tree[2*p+1].f);
tree[p].m = minPath( minPath(
                mergePath(tree[2*p].e, tree[2*p+1].f, M),
                mergePath(tree[2*p].f, tree[2*p+1].s, M)), minPath(
                mergePath(tree[2*p].f, tree[2*p+1].m, M),
                mergePath(tree[2*p].m, tree[2*p+1].f, M)));
tree[p].e = mergePath(tree[2*p].f, tree[2*p+1].e);
tree[p].f = mergePath(tree[2*p].f, tree[2*p+1].f);
```

이 때, leaf node 의 경우에는 특별한 주의가 필요한데 node 의 범위에 해당하는 장소가 1 개일 땐 s, m, e path 가 정의되지 않고 2 개일 때는 m path 가 정의되지 않기 때문이다. 따라서 update 를 할 때 terminal 조건으로는 f path 만을 update 하도록 해야 하고, 경로를 합치는 mergepath 를 수행할 때는 한쪽 끝이 정의되지 않은 path 일 경우에는 원래 path 그대로를 돌려 주어야 한다. 또한 M path 의 경우에는 양 끝이 아닌 장소를 패스할 수 있는 path 이기 때문에 양 끝 중 하나라도 정의가 되지 않았을 때는 path 자체를 정의할 수 없다. 이 예외 case 를 처리하기 위하여 위의 코드에서 m path 에 대한 mergePath 함수 호출에서는 M 을 argument 로 넘겨서 예외를 처리해야 함을 알려주도록 하였다.

위와 같은 방법으로 segment tree 를 만들고, 각 장소의 위치가 바뀔 때마다 수업 시간에 한 문제처럼 update 함수를 호출하여 tree 의 일부를 수정해 주면 된다. 또, query 역시 수업 시간에 한 문제처럼 각 node 별로 범위가 전혀 겹치지 않는지, 모두 겹치는지, 아니면 일부만 겹치는기를 고려하여 수동으로 합쳐 주면 된다. 여기에서 주의해야 할 점은 tree 의 자식 node 들 중 한 쪽의 범위만 필요로 하고 다른 한 쪽의 범위는 전혀 필요로 하지 않을 때 필요하지 않은 범위를 버려야만 한다는 것이다. 만약 그렇지 않게 되면 필요하지 않은 범위의 경우에는 정의되지 않은 path 들을 반환하기 때문에 merge 과정에서 모든 path 가 정의되지 않게 되어버린다.