

Introduction to Offline Queries



2020 Spring 중급



온라인 쿼리: 쿼리를 받는 대로 결과를 출력

오프라인 쿼리: 쿼리를 미리 전부 받아 놓고 결과를 나중에 한꺼번에 출력



온라인 쿼리: 쿼리를 받는 대로 결과를 출력

오프라인 쿼리: 쿼리를 미리 전부 받아 놓고 결과를 나중에 한꺼번에 출력

→ 굳이 이렇게 해야 되는 이유가 있을까?



트리에 다음 두 쿼리를 처리한다.

1. 어떤 정점과 그 부모 정점을 잇는 간선을 제거
2. 어떤 정점에서 다른 정점으로 갈 수 있는 경로가 있는지를 묻는 질의에 응답

노드 수, 쿼리 $\leq 200,000$.



쿼리를 순서대로 처리한다고 생각하면

1. 어떤 정점과 그 부모 정점을 잇는 간선을 제거
 - 그냥 제거해버리면 된다
2. 어떤 정점에서 다른 정점으로 갈 수 있는 경로가 있는지를 묻는 질의에 응답
 - ???

2번 쿼리의 경우 매번 DFS/BFS를 돌리는 것 외의 방법이 딱히 생각나지 않는데...



그래프에 간선을 동적으로 추가/제거하는 연산을 수행하는 문제를 동적 연결성 문제라고 한다

간선이 추가되기만 한다면?

- DSU^{disjoint set union}로 해결 가능!

간선이 제거될 수 있다면?

- 이건 가장 쉬운 문제가 5 ...



다행히도 아래와 같은 관찰을 통해 문제를 쉽게 바꿀 수 있는데

- 간선을 제거하는 것의 역연산은 간선을 추가하는 것이므로
- 쿼리가 들어오는 순서를 뒤집어 주면
- 간선을 추가하는 동적 연결성 문제가 되어 서로소 집합^{disjoint set} 문제와 같아진다!



이런 식으로

- 쿼리 하나하나를 처리하는 건 어려울 수 있지만
- 쿼리의 순서를 적절히 조작하면 문제가 쉬워지는 경우들이 있다!



지금까지 달성한 퀘스트들의 번호가 주어질 때, 다음 두 쿼리를 처리한다.

1. 퀘스트 번호 X 를 달성한다.
2. 퀘스트 번호 L 이상 R 이하인 퀘스트 중 달성하지 못한 퀘스트의 개수를 출력한다.

지금까지 달성한 퀘스트 $\leq 10^6$ 개, 쿼리 $\leq 10^6$ 개, $-10^9 \leq X, L, R \leq 10^9$.



지금까지 달성한 퀘스트 $\leq 10^6$ 개, 쿼리 $\leq 10^6$ 개, $-10^9 \leq X, L, R \leq 10^9$.

X, L, R 의 범위가 작았다면 세그먼트 트리로 해결할 수 있었을 텐데...



지금까지 달성한 퀘스트 $\leq 10^6$ 개, 쿼리 $\leq 10^6$ 개, $-10^9 \leq X, L, R \leq 10^9$.

X, L, R 의 범위가 작았다면 세그먼트 트리로 해결할 수 있었을 텐데...

그런데 어차피 쿼리가 1,000,000개니까

→ 등장할 수 있는 수의 종류는 최대 3,000,000개 아닌가?

(지금까지 달성한 퀘스트 수 + 쿼리 수 \times 2)



- 퀘스트 $1 \times 10^7, 3 \times 10^7, 10 \times 10^7, 14 \times 10^7$ 를 미리 달성했고
- 퀘스트 15×10^7 를 달성하는 쿼리
- 퀘스트 $-3 \times 10^7 \dots 10 \times 10^7$ 중에서 미해결 퀘스트 수를 출력하는 쿼리
- 퀘스트 -3×10^7 을 달성하는 쿼리
- 퀘스트 $-3 \times 10^7 \dots 7 \times 10^7$ 중에서 미해결 퀘스트 수를 출력하는 쿼리

를 순차적으로 처리해야 한다고 생각해보자



- 퀘스트 $1 \times 10^7, 3 \times 10^7, 10 \times 10^7, 14 \times 10^7$ 를 미리 달성했고
- 퀘스트 15×10^7 를 달성하는 쿼리
- 퀘스트 $-3 \times 10^7 \dots 10 \times 10^7$ 중에서 미해결 퀘스트 수를 출력하는 쿼리
- 퀘스트 -3×10^7 을 달성하는 쿼리
- 퀘스트 $-3 \times 10^7 \dots 7 \times 10^7$ 중에서 미해결 퀘스트 수를 출력하는 쿼리

위에서 등장하는 값들은

-3×10^7	1×10^7	3×10^7	7×10^7	10×10^7	14×10^7	15×10^7
------------------	-----------------	-----------------	-----------------	------------------	------------------	------------------

인데...



위에서 등장하는 값들은

-3×10^7	1×10^7	3×10^7	7×10^7	10×10^7	14×10^7	15×10^7
------------------	-----------------	-----------------	-----------------	------------------	------------------	------------------

이 값들을 '압축'하자



0	1	2	3	4	5	6
---	---	---	---	---	---	---



그렇게 하면 원래 퀴리들은

- 퀘스트 1, 2, 4, 5를 미리 달성했고
- 퀘스트 6을 달성하는 퀴리
- 퀘스트 0 ... 4 중에서 미해결 퀘스트 수를 출력하는 퀴리?
- 퀘스트 0을 달성하는 퀴리
- 퀘스트 0 ... 3 중에서 미해결 퀘스트 수를 출력하는 퀴리?

로 바뀐다



- 퀘스트 0 ... 4 중에서 미해결 퀘스트 수를 출력하는 쿼리?

이런 쿼리는 미해결 퀘스트 수를 출력하는 것이 아닌

(구간 길이) - (해결 퀘스트 수)

를 출력하는 것으로 취급

결과적으로 원소 3,000,000개의 세그먼트 트리를 관리하는 것으로 문제를 해결할 수 있다!

4 #15816 - 퀘스트 중인 모험가



-3×10^7	1×10^7	3×10^7	7×10^7	10×10^7	14×10^7	15×10^7
------------------	-----------------	-----------------	-----------------	------------------	------------------	------------------



0	1	2	3	4	5	6
---	---	---	---	---	---	---

```
vector<int> v;  
// ...  
sort(v.begin(), v.end());  
v.erase(unique(v.begin(), v.end()), v.end());
```

4 #15816 - 퀘스트 중인 모험가



-3×10^7	1×10^7	3×10^7	7×10^7	10×10^7	14×10^7	15×10^7
------------------	-----------------	-----------------	-----------------	------------------	------------------	------------------



0	1	2	3	4	5	6
---	---	---	---	---	---	---

- 특정 값을 압축한 인덱스가 얼마가 되는지는 이분 탐색으로 찾을 수 있다



```
vector<int> occur;

int n;
cin >> n;
for (int i = 0; i < n; i++) {
    cin >> initial_done[i];
    occur.emplace_back(initial_done[i]);
}
```

```
int q;
cin >> q;
for (int i = 0; i < q; i++) {
    cin >> queries[i].type >> queries[i].a;
    occur.emplace_back(queries[i].a);
    if (queries[i].type == 2) {
        cin >> queries[i].b;
        occur.emplace_back(queries[i].b);
    }
}
```

문제에서 등장하는 모든 값을 벡터 occur에 먼저 저장



```
sort(occur.begin(), occur.end());  
occur.erase(unique(occur.begin(), occur.end()), occur.end());
```

“값 압축”

퀘스트 중인 모험가

poj.kr/3506d9e10f184132bd3be545c58e4abe



```
segment_tree<int> segtree(v);
for (int i = 0; i < q; i++) {
    int op = queries[i].type;
    if (op == 1) {
        int a = lower_bound(occur.begin(), occur.end(), queries[i].a) - occur.begin();
        segtree.append(a, 1);
    } else {
        int a = lower_bound(occur.begin(), occur.end(), queries[i].a) - occur.begin();
        int b = lower_bound(occur.begin(), occur.end(), queries[i].b) - occur.begin();
        cout << (queries[i].b - queries[i].a + 1) - (segtree.query(a, b)) << '\n';
    }
}
```

원래 값을 압축된 값으로 바꾸는 작업은 lower_bound 등 이분 탐색을 이용해 가능



이렇게

- 원래 범위는 엄청나게 넓은데
- 등장하는 값의 수 자체가 적어서 원래 범위가 의미가 없고
- 등장하는 값의 수가 적음을 이용해 문제를 다른 방법으로 접근할 수 있게 된다면 등장하는 모든 값들을 미리 “압축”하는 방법을 활용할 수 있다



길이가 N 인 수열 A_1, A_2, \dots, A_N 이 주어진다. 이 때

- A_i, A_{i+1}, \dots, A_j 로 이루어진 부분 수열 중 k 보다 큰 원소의 개수를 구하는 쿼리를 수행하시오.

$N \leq 100,000$, 쿼리 수 $\leq 100,000$.



- A_i, A_{i+1}, \dots, A_j 로 이루어진 부분 수열 중 k 보다 큰 원소의 개수를 구하는 쿼리
→ 쿼리를 k 가 큰 순서로 정렬한다면 ...?
- A_1, A_2, \dots, A_N 들을 각각 “값 하나를 업데이트하는 쿼리”로 보고,
- 기존 쿼리와 섞어 k 가 큰 순서로 정렬한 것을 세그먼트 트리 등을 이용해 순서대로 처리
- 쿼리 출력 순서를 맞추기 위해 쿼리 결과를 다시 원래 순서대로 정렬해 준다



- Square root decomposition: 배열을 \sqrt{N} 개의 작은 배열로 쪼개서 관리
- Mo's: square root decomposition과 같이 쓰이는 알고리즘으로, 업데이트 쿼리가 없는 경우 구간의 범위와 길이에 따라 적절히 쿼리를 정렬해서 시간을 줄일 수 있다

이번 주의 문제 셋

문제 난이도는 2020. 5. 30 기준



5 #13306 – 트리

4 #15816 – 퀘스트 중인 모험가

4 #13537 – 수열과 쿼리

1 #17398 – 통신망 분할

5 #15586 – MooTube (Gold)

4 #2843 – 마블

4 #17469 – 트리의 색깔과 쿼리

3 #16978 – 수열과 쿼리 22

2 #15899 – 트리와 색깔

1 #5480 – 전함

5 #8452 – 그래프와 쿼리