2023 SCCC 봄 #8

BOJ 25288. 영어 시험

입력으로 주어진 문자열을 N번 출력하면 됩니다. 이것이 하한임은 자명하고, 상한임을 증명하는 것도 어렵지 않습니다.

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int N; string S; cin >> N >> S;
    for(int i=0; i<N; i++) cout << S;
}
```

BOJ 25343. 최장 최장 증가 부분 수열

최단 경로로 이동해야 하므로 오른쪽과 아래로만 이동할 수 있습니다.

$D(i,j) := (i,j)$를 마지막 원소로 하는 최장 증가 부분 수열의 길이라고 정의하면, 점화식은 왼쪽 위부터 차례대로 $D(i,j) = \max\limits_{r\le i, c\le j} D(r,c) + 1$를 이용해 채울 수 있습니다. 전체 시간 복잡도는 $O(N^4)$이고, 2차원 세그먼트 트리를 사용하면 $O(N^2 \log N)$에 해결할 수도 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, A[111][111], D[111][111], R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) for(int j=1; j<=N; j++) cin >> A[i][j];
    for(int i=1; i<=N; i++){
        for(int j=1; j<=N; j++){
            int mx = 0;
            for(int r=1; r<=i; r++) for(int c=1; c<=j; c++) if(A[r][c] < A[i]
[j]) mx = max(mx, D[r][c]);
            D[i][j] = mx + 1; R = max(R, D[i][j]);
        }
    }
    cout << R;
}
```

BOJ 25341. 인공 신경망

매번 인공 신경망을 통째로 계산하면 $O(NMQ)$가 되어서 시간 초과를 받게 됩니다. 연산 횟수를 줄여야 합니다.

잘 생각해 보면, 신경망 전체를 $O(NM)$ 시간에 입력이 N개인 인공 신경 하나로 합칠 수 있습니다. 신경망 전체를 인공 신경 하나로 합치면 매번 $O(N)$ 시간에 출력값을 계산할 수 있습니다. 따라서 전체 시간 복잡도는 $O(NM + NQ)$입니다.

여담으로, 이런 신경망은 선형 함수밖에 만들지 못하기 때문에 실제 신경망에서는 **활성화 함수**라는 것을 사용해 비선형 함수로 만듭니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll N, M, Q, A[5050], W[5050][5050], X[5050];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> Q;
    for(int i=1; i<=M; i++){
        int cnt; cin >> cnt;
        vector<int> idx(cnt);
        for(auto &j : idx) cin >> j;
        idx.push_back(N+1);
        for(int j=0; j<idx.size(); j++) cin >> W[i][idx[j]];
    }
    for(int i=1; i<=M; i++){
        ll t; cin >> t;
        for(int j=1; j<=N+1; j++) A[j] += t * W[i][j];
    }
    ll t; cin >> t; A[N+1] += t;

    for(int q=1; q<=Q; q++){
        for(int i=1; i<=N; i++) cin >> X[i]; X[N+1] = 1;
        ll res = 0;
        for(int i=1; i<=N+1; i++) res += A[i] * X[i];
        cout << res << "\n";
    }
}
```

BOJ 20530. 양분

$M = N - 1$이면 트리이고, 트리에서는 임의의 두 정점을 연결하는 경로가 정확히 1개 존재합니다.

$M = N$이면 트리에 간선 하나를 추가한 그래프이고, 이 그래프는 사이클 하나와 주변에 트리 형태의 가지 여러 개로 구성되어 있습니다. 만약 동일한 트리 안에서 이동한다면 여전히 경로는 1개지만, 그렇지 않은 경우에는 사이클을 돌아가는 방법이 2가지이므로 경로가 2개 존재합니다.

사이클과 트리 부분을 잘 분리해야 합니다.

그래프에서 차수가 1인 정점을 계속 지워나가면 마지막에 사이클 하나만 남게 됩니다. 또한, 이 과정에서 지워지는 정점마다 가장 가까운 사이클 위의 점을 구하면 각 정점이 어떤 트리에 속하는 지 구할 수 있습니다. 따라서 위상 정렬과 비슷한 방식으로 $O(N)$ 시간에 트리를 떼어낼 수 있습니다.

쿼리를 처리하는 것은 주어진 두 점점이 같은 트리에 속한다면 1, 그렇지 않으면 2를 출력하면 됩니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, Q, Degree[202020], TreeID[202020];
vector<int> G[202020];
bool IsCycle[202020];

void GetCycle(){
```

```
    queue<int> q;
    for(int i=1; i<=N; i++) if(!--Degree[i]) q.push(i);
    while(q.size()){
        int v = q.front(); q.pop();
        for(auto i : G[v]) if(!--Degree[i]) q.push(i);
    }
    for(int i=1; i<=N; i++) IsCycle[i] = Degree[i] > 0;
}

void GetTree(int v, int st){
    TreeID[v] = st;
    for(auto i : G[v]) if(!TreeID[i] && !IsCycle[i]) GetTree(i, st);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> Q;
    for(int i=1; i<=N; i++){
        int s, e; cin >> s >> e;
        G[s].push_back(e);
        G[e].push_back(s);
        Degree[s]++; Degree[e]++;
    }
    GetCycle();
    for(int i=1; i<=N; i++) if(IsCycle[i]) GetTree(i, i);

    for(int i=1; i<=Q; i++){
        int u, v; cin  >> u >> v;
        if(TreeID[u] && TreeID[v] && TreeID[u] == TreeID[v]) cout << 1 << "\n";
        else cout << 2 << "\n";
    }
}
```

BOJ 27381. 순찰 경로

주어진 트리가 성게 그래프이면 불가능하고, 그렇지 않으면 항상 가능합니다. 경로를 구성하는 방법으로 센트로이드를 이용한 풀이와 이분 그래프를 이용한 풀이가 존재하는데, 차례대로 소개하겠습니다.

시작 정점 s를 고정합시다. 트리는 s를 기준으로 몇 개의 서브트리로 나누어 집니다. 매번 이전에 방문한 서브트리와 다른 서브트리로 이동하면 올바른 경로를 만들 수 있지만, 만약 어떤 서브트리가 너무 크다면 매번 다른 서브트리로 이동하지 못할 수도 있습니다. 시작 정점에 따라 경로를 찾지 못할 수도 있기 때문에 시작 정점을 잘 선택해야 합니다.

만약 모든 서브트리의 크기가 $N/2$ 이하라면 항상 올바른 경로를 찾을 수 있습니다. 모든 서브트리의 크기를 $N/2$ 이하로 만드는 정점은 **센트로이드**라는 이름으로 잘 알려져 있고, $O(N)$ 시간에 구할 수 있습니다. 따라서 센트로이드에서 시작해서 매번 다른 서브트리로 이동하면 문제를 해결할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, K, D[202020], S[202020];
vector<int> G[202020];

void Clear(){
    memset(D+1, 0, sizeof(D[0]) * N);
    memset(S+1, 0, sizeof(S[0]) * N);
    for(int i=1; i<=N; i++) G[i].clear();
```

```
}

void AddEdge(int u, int v){
    G[u].push_back(v); D[u]++;
    G[v].push_back(u); D[v]++;
}

int GetSize(int v, int p){
    S[v] = 1;
    for(auto i : G[v]) if(i != p) S[v] += GetSize(i, v);
    return S[v];
}

int GetCent(int v, int p){
    for(auto i : G[v]) if(i != p && S[i] * 2 > N) return GetCent(i, v);
    return v;
}

vector<int> Gather(int cent, int st){
    vector<int> res;
    function<void(int,int)> dfs = [&](int v, int p){
        res.push_back(v);
        for(auto i : G[v]) if(i != p) dfs(i, v);
    };
    dfs(st, cent);
    return res;
}

void Solve(){
    cin >> N;
    for(int i=1,u,v; i<N; i++) cin >> u >> v, AddEdge(u, v);
    if(*max_element(D+1, D+N+1) + 1 == N){ cout << "-1\n"; Clear(); return; }
    GetSize(1, -1); K = GetCent(1, -1);

    vector<vector<int>> V;
    for(auto i : G[K]) V.push_back(Gather(K, i));

    priority_queue<pair<int,int>> Q;
    for(int i=0; i<V.size(); i++) Q.emplace(V[i].size(), i);

    cout << K << " ";
    while(!Q.empty()){
        if(Q.size() == 1){
            auto [s,v] = Q.top(); Q.pop();
            cout << V[v].back() << " "; V[v].pop_back();
            continue;
        }
        auto [s1,v1] = Q.top(); Q.pop();
        auto [s2,v2] = Q.top(); Q.pop();
        cout << V[v1].back() << " "; V[v1].pop_back();
        cout << V[v2].back() << " "; V[v2].pop_back();
        if(--s1 > 0) Q.emplace(s1, v1);
        if(--s2 > 0) Q.emplace(s2, v2);
    }
    cout << "\n";
    Clear();
}
```

```
int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int TC; cin >> TC;
    for(int tc=1; tc<=TC; tc++) Solve();
}
```

트리에서 깊이가 홀수인 정점과 짝수인 정점은 각자 서로 연결되어 있지 않습니다. 따라서 깊이가 홀수인 정점을 모두 방문한 다음 짝수인 정점을 방문하면 문제를 해결할 수 있습니다.

이때 깊이가 홀수인 정점에서 짝수인 정점으로 넘어가는 부분을 잘 처리해야 하는데, 깊이가 $2, 4, \cdots,$인 정점을 순서대로 모두 방문한 다음, 깊이가 $1, 3, \cdots$인 정점을 방문하면 됩니다. 이때 루트에서 가장 먼 정점까지의 거리가 3 이상이 되어야 하는데, 성게 그래프가 아닌 경우에는 임의의 리프 정점을 루트로 잡으면 됩니다.

```
#include <bits/stdc++.h>
using namespace std;

int N;
vector<vector<int>> G, V;

void DFS(int v, int b=-1, int d=1){
    V[d].push_back(v);
    for(auto i : G[v]) if(i != b) DFS(i, v, d+1);
}

void Solve(){
    cin >> N; G = V = vector<vector<int>>(N+1);
    for(int i=1,u,v; i<N; i++) cin >> u >> v, G[u].push_back(v),
G[v].push_back(u);
    for(int i=1; i<=N; i++) if(G[i].size() + 1 == N) { cout << "-1\n"; return; }
    for(int i=1; i<=N; i++) if(G[i].size() == 1) { DFS(i); break; }
    for(int i=2; i<=N; i+=2) for(auto j : V[i]) cout << j << " ";
    for(int i=1; i<=N; i+=2) for(auto j : V[i]) cout << j << " ";
    cout << "\n";
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int TC; cin >> TC;
    for(int tc=1; tc<=TC; tc++) Solve();
}
```

BOJ 22997. 미사일 폭격

$O((N+Q)\sqrt{N}\log^2 X)$ 풀이와 $O(q\log^2 X)$ 풀이가 있습니다. 차례대로 소개합니다.

i번째 부대가 존재하는 시간 구간 $[s_i, e_i]$를 생각해 보면, 해당 시간에 (x_i, y_i)가 피해를 입었는지 판별하는 문제가 됩니다. Mo's Algorithm으로 해결할 수 있습니다.

미사일 폭격을 구간에 넣는 것은 직사각형 영역에 1을 더하는 연산, 구간에서 빼는 것은 직사각형 영역에 1을 빼는 연산이라고 생각할 수 있으므로 2D Fenwick Tree를 이용해 구간의 이동을 매번 $O(\log^2 X)$에 처리할 수 있습니다. Mo's Algorithm에서 구간의 끝점은 최대 $O((N+Q)\sqrt{N})$ 움직이므로 전체 시간 복잡도는 $O((N+Q)\sqrt{N}\log^2 X)$입니다.

```
#include <bits/stdc++.h>
using namespace std;
```

```cpp
constexpr int MAX_N = 202020, MAX_M = 202020, MAX_K = 202020, MAX_Q = 404040,
MAX_X = 505;
constexpr int MAX_SZ = 1 << 9, BUCKET = 500, INS = +1, DEL = -1;

template< size_t _Sz = MAX_SZ<<2 >
struct FenwickTree{
    int T[_Sz][_Sz];
    void add(int x, int y, int v){
        for(x+=2; x<_Sz; x+=x&-x) for(int yy=y+2; yy<_Sz; yy+=yy&-yy) T[x][yy]
+= v;
    }
    int get(int x, int y){
        int ret = 0;
        for(x+=2; x; x-=x&-x) for(int yy=y+2; yy; yy-=yy&-yy) ret += T[x][yy];
        return ret;
    }
};

struct Query{
    int s, e, x, buc;
    Query() = default;
    Query(int s, int e, int x) : s(s), e(e), x(x), buc(s / BUCKET) {}
    bool operator < (const Query &t) const {
        if(buc != t.buc) return buc < t.buc;
        if(buc & 1) return e < t.e;
        return e > t.e;
    }
};

int N, M, K, Q;
int Type[MAX_Q], X[MAX_Q], Y[MAX_Q], R[MAX_Q], Idx[MAX_Q], InsTime[MAX_N],
DelTime[MAX_N], Counter;
int L1[MAX_Q], R1[MAX_Q], L2[MAX_Q], R2[MAX_Q];
vector<Query> Queries;
FenwickTree<> Tree;

void Update(int r1, int c1, int r2, int c2, int v){
    Tree.add(r1, c1, v);
    Tree.add(r1, c2+1, -v);
    Tree.add(r2+1, c1, -v);
    Tree.add(r2+1, c2+1, v);
}
int Query(int r, int c){
    return Tree.get(r, c);
}
void Mo(int op, int ti){
    if(Type[ti] == 1) Update(L1[ti], R1[ti], L2[ti], R2[ti], op);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> K; Q = M + K;
    for(int i=1; i<=Q; i++){
        cin >> Type[i];
        if(Type[i] == 1) cin >> X[i] >> Y[i] >> R[i];
        if(Type[i] == 2) cin >> X[i] >> Y[i], Idx[i] = ++Counter,
InsTime[Idx[i]] = i;
        if(Type[i] == 3) cin >> Idx[i], DelTime[Idx[i]] = i;
```

```
        int x = X[i], y = Y[i];
        X[i] = x + y + MAX_X;
        Y[i] = x - y + MAX_X * 2;
        if(Type[i] == 1){
            L1[i] = X[i] - R[i]; R1[i] = Y[i] - R[i];
            L2[i] = X[i] + R[i], R2[i] = Y[i] + R[i];
        }
    }
    for(int i=1; i<=N; i++){
        if(!InsTime[i]) continue;
        if(!DelTime[i]) DelTime[i] = Q + 1;
        Queries.emplace_back(InsTime[i], DelTime[i], i);
    }
    sort(Queries.begin(), Queries.end());

    int s = Queries[0].s, e = Queries[0].e, res = 0;
    for(int i=s; i<=e; i++) Mo(+1, i);
    if(Query(X[s], Y[s])) res++;

    for(int i=1; i<Queries.size(); i++){
        while(Queries[i].s < s) Mo(INS, --s);
        while(e < Queries[i].e) Mo(INS, ++e);
        while(s < Queries[i].s) Mo(DEL, s++);
        while(Queries[i].e < e) Mo(DEL, e--);
        if(Query(X[s], Y[s])) res++;
    }
    cout << res;
}
```

Mo's Algorithm을 이용한 풀이에서의 아이디어를 그대로 사용합니다.

시간 구간 $[s_i, e_i]$ 동안 (x_i, y_i)가 피해를 입은 적이 있는지 계산해야 합니다. 지금까지 (x_i, y_i)가 미사일에 몇 번 피해를 받았는지 구할 수 있다면, s_i 시점과 e_i 시점까지 각각 (x_i, y_i)가 미사일에 피해를 입은 횟수를 구할 수 있습니다. 두 값이 다른지 확인하면 문제를 해결할 수 있습니다.

이를 위해서는 모든 미사일을 시간 순으로 보면서, 미사일이 떨어질 때마다 직사각형 영역에 어떤 수를 더하는 쿼리와 특정 지점의 값을 구하는 쿼리를 처리해야 합니다. 2차원 펜윅 트리를 이용해 매번 $O(\log^2 X)$ 시간에 처리할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

template< size_t _Sz>
struct FenwickTree{
    int T[_Sz][_Sz];
    void add(int x, int y, int v){
        for(x+=3; x<_Sz; x+=x&-x) for(int yy=y+3; yy<_Sz; yy+=yy&-yy) T[x][yy]
+= v;
    }
    int get(int x, int y){
        int ret = 0;
        for(x+=3; x; x-=x&-x) for(int yy=y+3; yy; yy-=yy&-yy) ret += T[x][yy];
        return ret;
    }
};
```

```
int N, M, K, Q, X[202020], Y[202020], V[202020], Dead[202020], ID, Res;
FenwickTree<1024> T;

void Update(int sx, int sy, int ex, int ey, int v){
    T.add(sx, sy, v);
    T.add(sx, ey+1, -v);
    T.add(ex+1, sy, -v);
    T.add(ex+1, ey+1, v);
}
int Query(int x, int y){
    return T.get(x, y);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> K; Q = M + K;
    for(int i=1; i<=Q; i++){
        int op; cin >> op;
        if(op == 1){
            int _x, _y, x, y, r;
            cin >> _x >> _y >> r;
            x = _x + _y; y = _x - _y + 500;
            Update(max(1, x-r), max(1, y-r), min(1000, x+r), min(1000, y+r), 1);
        }
        else if(op == 2){
            int _x, _y, x, y;
            cin >> _x >> _y;
            x = _x + _y; y = _x - _y + 500;
            ID++;
            X[ID] = x; Y[ID] = y;
            V[ID] = Query(x, y);
        }
        else{
            int k; cin >> k;
            if(Query(X[k], Y[k]) > V[k]) Res++;
            Dead[k] = 1;
        }
    }

    for(int i=1; i<=ID; i++){
        if(!Dead[i] && Query(X[i], Y[i]) > V[i]) Res++;
    }

    cout << Res;
}
```

BOJ 17668. 시험

각 학생을 3차원 위의 점 $(X_i, Y_i, X_i + Y_i)$라고 생각하면, $[0, X_j] \times [0, Y_j] \times [0, Z_j]$ 영역에 포함된 점 개수를 세는 문제라고 생각할 수 있습니다.

2차원이라면 점과 쿼리를 모두 모아서 (x, y) pair로 정렬한 다음, x좌표가 작은 점부터 차례대로 보면서 y좌표를 기준으로 세그먼트 트리에 쿼리를 하는 방식으로 해결할 수 있습니다. 정렬을 통해 x좌표에 대한 기준을 없애고, 세그먼트 트리의 인덱스를 통해 자연스럽게 y좌표 조건을 충족시키는 방식입니다. 3차원에서는 어떻게 해결해야 할까요?

정 모르겠으면 일단 반으로 쪼개서 분할 정복을 생각해 봐야 합니다. 앞쪽 절반은 쿼리를 처리하지 않고 점을 넣기만 할 것이고, 뒤쪽 절반은 점을 넣지 않고 쿼리만 처리합니다. 이렇게 하면 추가된 차원에 대한 조건도 충족시킬 수 있습니다.

구체적으로, 점들을 x좌표 기준으로 정렬한 다음 분할 정복을 합니다. 점과 쿼리가 모두 한쪽에 몰려있는 경우는 재귀적으로 처리합니다. 정복 과정에서는 점이 왼쪽 절반, 쿼리가 오른쪽 절반에 있는 경우만 고려합니다. 모든 쿼리의 x좌표가 점의 x좌표보다 크기 때문에, 정복 과정은 축이 2개밖에 없는 2차원 문제라고 생각할 수 있습니다. 따라서 정복 과정은 $O((N+Q)\log(N+Q))$ 시간에 해결할 수 있고, 전체 시간 복잡도는 $O((N+Q)\log^2(N+Q))$가 됩니다.

```
#include <bits/stdc++.h>
#define all(v) v.begin(), v.end()
#define compress(v) sort(all(v)), v.erase(unique(all(v)), v.end())
#define IDX(v, x) lower_bound(all(v), x) - v.begin()
using namespace std;
using ll = long long;

struct Point3D{ int x, y, z, i; };
template<int _Sz> struct FenwickTree{
    int T[_Sz];
    void Update(int x, int v){ for(x+=2; x<_Sz; x+=x&-x) T[x] += v; }
    int Query(int x){ int ret = 0; for(x+=2; x; x-=x&-x) ret += T[x]; return
ret; }
    int Query(int s, int e){ return Query(e) - Query(s-1); }
};

bool Compare(const Point3D &a, const Point3D &b){
    return make_tuple(-a.x, a.y, a.z, a.i) < make_tuple(-b.x, b.y, b.z, b.i);
}

int N, Q, Ans[101010];
FenwickTree<202020> bit;
Point3D A[202020], Temp[202020];
vector<int> Z_comp, vec;

void Solve(int s, int e){
    if(s >= e) return;
    int m = s + e >> 1;
    Solve(s, m); Solve(m+1, e);
    vec.clear();
    int i = s, j = m+1, idx = s, cnt = 0;
    while(i <= m || j <= e){
        if(j > e || (i <= m && A[j].y < A[i].y)){
            if(!A[i].i) bit.Update(A[i].z, 1), vec.push_back(A[i].z), cnt++;
            Temp[idx++] = A[i++];
        }
        else{
            if(A[j].i) Ans[A[j].i] += cnt - bit.Query(0, A[j].z);
            Temp[idx++] = A[j++];
        }
    }
    memcpy(A+s, Temp+s, sizeof(Point3D)*(e-s+1));
    for(const auto &k : vec) bit.Update(k, -1);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
```

```
    cin >> N >> Q; Z_comp.reserve(N+Q); vec.reserve(N);
    for(int i=1; i<=N; i++){
        cin >> A[i].x >> A[i].y;
        A[i].z = A[i].x + A[i].y;
        Z_comp.push_back(A[i].z);
    }
    for(int i=N+1; i<=N+Q; i++){
        cin >> A[i].x >> A[i].y >> A[i].z;
        A[i].x--; A[i].y--; A[i].z--;
        A[i].i = i-N;
        Z_comp.push_back(A[i].z);
    }
    compress(Z_comp);
    for(int i=1; i<=N+Q; i++) A[i].z = IDX(Z_comp, A[i].z);
    sort(A+1, A+N+Q+1, Compare);
    Solve(1, N+Q);
    for(int i=1; i<=Q; i++) cout << Ans[i] << "\n";
}
```

BOJ 19619. 자매 도시

MST는 다음과 같은 성질을 갖고 있습니다.

- 경로 위에 있는 가중치의 최댓값을 최소화

이 문제에서 요구하는 쿼리와 MST의 성질이 잘 맞는 것 같으니 MST 위주로 풀이를 생각해 봅시다.

두 자동차가 위치를 교환할 수 있는 경우에 그래프가 어떤 형태인지, 그리고 어떤 방식으로 이동하는지 알아봅시다.

- degree가 3 이상인 정점 x가 존재
  - 첫 번째 차가 x까지 이동한 다음, 한 칸 더 가서 **주차**함
  - 두 번째 차가 x까지 이동한 다음, 도착 지점으로 이동
  - **주차**한 상태였던 첫 번째 차가 다시 출발해서 도착 지점으로 이동
- 사이클이 존재
  - 두 차 모두 사이클로 들어간 뒤, 사이클을 돌면서 위치를 교환

두 가지 경우에 모두 속하지 않는 그래프는 선형 그래프밖에 없습니다. 즉, 선형 그래프가 아니라면 항상 두 자동차가 위치를 교환할 수 있습니다.

두 자동차가 위치를 교환할 수 있는 상태를 나타내는 새로운 정점 X를 만듭시다. 두 자동차 U, V가 위치를 교환하는 비용은 max(U에서 X로 이동하는 비용 + V에서 X로 이동하는 비용, U에서 V로 이동하는 비용)입니다

X를 포함해서 MST를 구축하면 Sparse Table이나 HLD+Segment Tree를 이용해서 문제를 풀 수 있습니다.

두 자동차가 위치를 교환할 수 있는 조건은 (1) degree가 3 이상인 정점 x에 도달함, (2) 사이클에 도달함, 총 2가지입니다.

원래 주어진 그래프에서 크루스칼 알고리즘을 이용해 MST를 만들면서

1. 차수가 3 이상이 된 정점 Y에 대해 X와 Y를 잇는 간선을 추가하고
2. 어떤 간선이 잇는 두 정점 Y_1, Y_2가 이미 같은 컴포넌트에 속한 경우, X와 Y_1을 잇는 간선과 X와 Y_2를 잇는 간선을 각각 추가하면 됩니다.

```
// #include "swap.h"
```

```cpp
#include <bits/stdc++.h>
using namespace std;
constexpr int SZ = 1 << 17;

struct Edge{
    int u, v, w;
    Edge() = default;
    Edge(int u, int v, int w) : u(u), v(v), w(w) {}
    bool operator < (const Edge &e) const { return w < e.w; }
};

struct UnionFind{
    int P[SZ];
    UnionFind(){ clear(); }
    void clear(){ iota(P, P+SZ, 0); }
    int find(int v){ return v == P[v] ? v : P[v] = find(P[v]); }
    bool merge(int u, int v){ return find(u) != find(v) && (P[P[u]]=P[v], true);
}
};

struct SegmentTree{
    int T[SZ<<1];
    SegmentTree(){ clear(); }
    void clear(){ memset(T, 0, sizeof T); }
    void update(int x, int v){
        for(T[x|=SZ]=v; x>>=1; ) T[x] = max(T[x<<1], T[x<<1|1]);
    }
    int query(int l, int r){
        int res = 0;
        for(l|=SZ, r|=SZ; l<=r; l>>=1, r>>=1){
            if(l & 1) res = max(res, T[l++]);
            if(~r & 1) res = max(res, T[r--]);
        }
        return res;
    }
};

struct HLD{
    vector<pair<int,int>> inp[SZ];
    vector<int> g[SZ]; SegmentTree tree;
    int in[SZ], top[SZ], dep[SZ], sz[SZ], par[SZ], w[SZ], pv;
    void add_edge(int u, int v, int w){ inp[u].emplace_back(v, w);
inp[v].emplace_back(u, w); }
    void dfs0(int v, int b=-1){
        for(auto [i,c] : inp[v]) if(i != b) g[v].push_back(i), w[i] = c, dep[i]
= dep[v] + 1, par[i] = v, dfs0(i, v);
    }
    void dfs1(int v){
        sz[v] = 1;
        for(auto &i : g[v]){
            dfs1(i); sz[v] += sz[i];
            if(sz[i] > sz[g[v][0]]) swap(i, g[v][0]);
        }
    }
    void dfs2(int v){
        in[v] = ++pv;
        for(auto i : g[v]) top[i] = i == g[v][0] ? top[v] : i, dfs2(i);
    }
```

```
    void init(int n){
        tree.clear();
        pv = 0; dfs0(0); dfs1(0); dfs2(top[0]=0);
        for(int i=0; i<n; i++) tree.update(in[i], w[i]);
    }
    int query(int u, int v){
        int res = 0;
        for(; top[u] != top[v]; u=par[top[u]]){
            if(dep[top[u]] < dep[top[v]]) swap(u, v);
            res = max(res, tree.query(in[top[u]], in[u]));
        }
        if(dep[u] > dep[v]) swap(u, v);
        return max(res, tree.query(in[u]+1, in[v]));
    }
};

int N, M, Deg[101010], Flag;
vector<Edge> E;
UnionFind U; HLD T1, T2;

void init(int n, int m, vector<int> u, vector<int> v, vector<int> w) {
    N = n; M = m; E.reserve(M);
    for(int i=0; i<M; i++) E.emplace_back(u[i], v[i], w[i]);
    sort(E.begin(), E.end());

    U.clear();
    for(auto e : E) if(U.merge(e.u, e.v)) T1.add_edge(e.u, e.v, e.w);

    U.clear();
    for(auto e : E){
        if(U.merge(e.u, e.v)) T2.add_edge(e.u, e.v, e.w);
        else{
            if(U.merge(e.u, n)) T2.add_edge(e.u, n, e.w);
            if(U.merge(e.v, n)) T2.add_edge(e.v, n, e.w);
        }
        if(++Deg[e.u] == 3 && U.merge(e.u, n)) T2.add_edge(e.u, n, e.w), Flag =
1;
        if(++Deg[e.v] == 3 && U.merge(e.v, n)) T2.add_edge(e.v, n, e.w), Flag =
1;
    }
    if(N <= M) Flag = 1;
    T1.init(n); T2.init(n+1);
}

int getMinimumFuelCapacity(int a, int b){
    if(!Flag) return -1;
    return max({T1.query(a, b), T2.query(a, b), T2.query(a, N), T2.query(b,
N)});
}
```

BOJ 2586. 소방차

최적해가 어떤 형태인지, 혹은 특정한 형태를 강제할 수 있는지 생각해 봅시다.

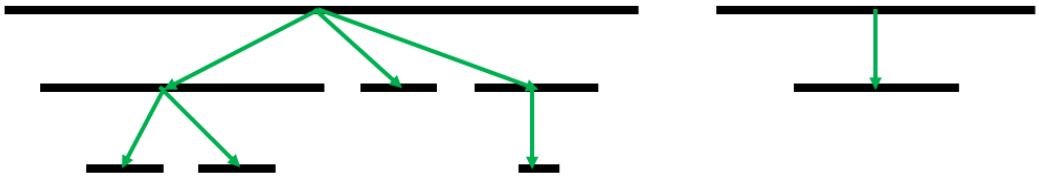
소방차와 펌프를 매칭하는 것을 **구간**이라고 생각하면, 구간들이 서로를 **완전히 포함**하거나 **완전히 분리**된 형태의 최적해가 존재합니다.

Proof) $a \le b \le c \le d$ 일 때 $[a, c]$, $[b, d]$를 매칭하는 것이 최적이라면 $[a, d]$, $[b, c]$로 바꿔도 거리는 동일합니다.

최적해에 $[l, r]$ 매칭이 존재한다면, 이 구간 안에 있는 소방차와 펌프는 모두 매칭되어 있습니다.

Proof) 그렇지 않으면 적당히 바꿔서 거리를 더 줄일 수 있습니다.

구간의 포함 관계를 트리(or 포레스트)로 생각할 수 있습니다. 이때 같은 depth에서는 모든 구간이 분리되어 있습니다.



펌프와 소방차의 depth는 아래 과정을 통해 결정할 수 있습니다.

1. 입력으로 들어온 모든 좌표를 정렬
2. 좌표 순서대로 보면서, 펌프/소방차에 따라 다음 과정을 수행

- 펌프가 나오면 현재 depth에 넣고 depth++
- 소방차가 나오면 --depth하고 depth에 넣음

같은 depth에서는 펌프와 소방차가 교대로 등장합니다. 따라서 어떤 depth에 원소가 짝수 개 있다면 순서대로 매칭하면 됩니다. 홀수인 경우에는 하나를 빼고 매칭하는 K가지 경우를 $O(K)$만에 모두 확인하면 됩니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int N, M, K; ll R;
pair<int,int> A[202020];
vector<pair<int,int>> V[404040];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M; K = N + M;
    for(int i=1; i<=N; i++) cin >> A[i].first, A[i].second = 0;
    for(int i=1; i<=M; i++) cin >> A[N+i].first, A[N+i].second = 1;
    sort(A+1, A+K+1);

    int C = K;
    for(int i=1; i<=K; i++){
        if(A[i].second == 0) V[C++].push_back(A[i]);
        else V[--C].push_back(A[i]);
    }
    for(int i=0; i<404040; i++){
        if(V[i].empty()) continue;
        ll now = 0;
```

```
        for(int j=1; j<V[i].size(); j+=2) now += abs(V[i][j].first - V[i][j-
1].first);
        if(V[i].size() % 2 == 0){ R += now; continue; }
        ll mn = now;
        for(int j=(int)V[i].size()-1; j>=2; j-=2){
            now += abs(V[i][j].first - V[i][j-1].first) - abs(V[i][j-1].first -
V[i][j-2].first);
            mn = min(mn, now);
        }
        R += mn;
    }
    cout << R;
}
```