2023 SCCC 봄 #7

BOJ 21967. 세워라 반석 위에

수열의 길이를 N, 최댓값을 X라고 합시다. 구간의 최댓값과 최솟값의 차이가 2 이하인 가장 긴 구간을 구해야 합니다.

X가 매우 작은 것에 주목해야 합니다. 임의의 자연수 l에 대해, $l \le A_i \le l+2$인 구간의 최대 길이는 $O(N)$ 시간에 구할 수 있습니다.

$X-2$개의 l만 확인하면 되므로 $O(NX)$에 문제를 해결할 수 있습니다.

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

int N, A[1010101];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];

    int mx = 0;
    for(int i=1; i+2<=10; i++){
        int now = 0;
        for(int j=1; j<=N; j++){
            if(i <= A[j] && A[j] <= i+2) mx = max(mx, ++now);
            else now = 0;
        }
    }
    cout << mx;
}
```

BOJ 27972. 악보는 거들 뿐

수가 증가하는 연속한 부분 수열에서는 매번 증가하는 수열을 작성해야 하고, 감소하는 연속 부분 수열에서는 감소하는 수열을 작성해야 합니다. 작성하는 수의 최댓값을 최소화해야 하므로, 증가하는 구간은 $1, 2, 3, \cdots$으로, 감소하는 구간은 $\cdots, 3, 2, 1$로 작성하면 됩니다.

따라서 증가하는 구간의 최대 길이와 감소하는 구간의 최대 길이 중 더 큰 값이 정답이 됩니다.

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

int N, A[505050], R=1;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];

    int now = 1;
```

```
    for(int i=2; i<=N; i++){
        if(A[i-1] == A[i]) continue;
        if(A[i-1] < A[i]) now++;
        else now = 1;
        R = max(R, now);
    }
    reverse(A+1, A+N+1);
    now = 1;
    for(int i=2; i<=N; i++){
        if(A[i-1] == A[i]) continue;
        if(A[i-1] < A[i]) now++;
        else now = 1;
        R = max(R, now);
    }
    cout << R;
}
```

BOJ 22967. 구름다리

정답이 1이 되는 경우를 먼저 생각해 보면, 정답이 1인 것은 간선이 $N(N-1)/2$개 있는 것과 동치입니다. 이미 간선이 $N-1$개 있고, 간선을 $N-1$개 더 추가할 수 있으므로 $2(N-1) \geq N(N-1)/2$일 때만 정답이 1이 될 수 있습니다. 따라서 $N \leq 4$이면 완전 그래프를 만들어서 정답을 1로 만들 수 있습니다.

$N > 4$일 때는 1번 정점과 $2, 3, \cdots, N$번 정점을 연결하면 항상 정답을 2로 만들 수 있습니다.

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

int N, G[333][333];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<N; i++){
        int u, v; cin >> u >> v;
        G[u][v] = G[v][u] = 1;
    }

    vector<pair<int, int>> Res;
    if(N <= 4){
        for(int i=1; i<=N; i++){
            for(int j=i+1; j<=N; j++){
                if(!G[i][j]) Res.emplace_back(i, j);
            }
        }
    }
    else{
        for(int i=2; i<=N; i++){
            if(!G[1][i]) Res.emplace_back(1, i);
        }
    }

    cout << Res.size() << "\n";
    cout << (N <= 4 ? 1 : 2) << "\n";
    for(auto i : Res) cout << i.first << " " << i.second << "\n";
}
```

BOJ 12975. 트라이슬

가장 먼저 생각할 수 있는 풀이는 다음과 같습니다.

- $D(i, a, b, c, flag) := i$번째 사람까지 고려했을 때, 세 팀의 힘이 각각 a, b, c이고, 각 팀에 1명 이상 들어갔는지 여부가 $flag$일 수 있으면 1, 불가능하면 0

하지만 이 점화식은 상태가 $256^3 \times 8 \times N$가지이므로 제한 시간 안에 정답을 구할 수 없습니다.

a, b가 결정되면 c는 자동으로 결정된다는 점을 이용하면 차원을 하나 줄일 수 있고, 상태의 개수가 $8 \times 256^2 \times N \leq\approx 5 \times 10^7$개로 줄어들기 때문에 문제를 해결할 수 있습니다.

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

int N, A[111], D[111][256][256][8], S;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i], S ^= A[i];
    D[0][0][0][0] = 1;
    for(int i=1; i<=N; i++){
        for(int j=0; j<256; j++){
            for(int k=0; k<256; k++){
                for(int s=0; s<8; s++){
                    D[i][j][k][s|1] |= D[i-1][j][k][s];
                    D[i][j][k][s|2] |= D[i-1][j^A[i]][k][s];
                    D[i][j][k][s|4] |= D[i-1][j][k^A[i]][s];
                }
            }
        }
    }

    int R = 0;
    for(int i=0; i<256; i++) for(int j=0; j<256; j++) if(D[N][i][j][7]) R =
max(R, i + j + (S ^ i ^ j));
    cout << R;
}
```

BOJ 27471. 시그마 시그마 시그마 시그마

시그마가 4개 중첩되어 있지만 별로 복잡하지 않습니다. 길이가 2 이상인 모든 연속한 부분 수열에서, 가능한 모든 원소 쌍에 대해 둘 중 최댓값을 더하는 문제입니다. 즉, A의 연속한 부분 수열 $A[l \cdots r]$에서, 모든 $l \leq i < j \, leqr$에 대해 $\max(A_i, A_j)$의 합을 구하는 문제입니다.

$A_i = A_j$인 경우를 처리하는 것이 귀찮을 것 같으니 수를 비교하는 대신 순서쌍 (A_i, i)와 (A_j, j)를 비교합시다. 이렇게 하면 값이 같더라도 최대가 되는 원소를 항상 일정한 방향으로 고정할 수 있습니다.

더블 카운팅과 비슷한 느낌으로 A_i가 최대가 되는 경우의 수를 구합시다. A_i의 왼쪽에서 A_i 이하인 값의 개수, 그리고 A_i의 오른쪽에 있으면서 A_i 미만인 값의 개수를 구하면 됩니다. 세그먼트 트리를 이용해 $O(N \log N)$ 시간에 계산할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int SZ = 1 << 19;
```

```
constexpr ll MOD = 998244353;

struct Tree{
    ll T[SZ<<1];
    void update(int x, ll v){
        x |= SZ; T[x] = v % MOD;
        while(x >>= 1) T[x] = (T[x<<1] + T[x<<1|1]) % MOD;
    }
    ll query(int l, int r){
        l |= SZ; r |= SZ; ll res = 0;
        while(l <= r){
            if(l & 1) res = (res + T[l++]) % MOD;
            if(~r & 1) res = (res + T[r--]) % MOD;
            l >>= 1; r >>= 1;
        }
        return res;
    }
} T1, T2;

ll N, A[303030], R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i], T1.update(i, i), T2.update(i, N-i+1);

    vector<pair<ll,ll>> V;
    for(int i=1; i<=N; i++) V.emplace_back(A[i], i);
    sort(V.begin(), V.end(), greater<>());

    for(auto [v,i] : V){
        T1.update(i, 0); T2.update(i, 0);
        R += T1.query(1, i-1) * v % MOD * (N-i+1) % MOD;
        R += T2.query(i+1, N) * v % MOD * i % MOD;
        R %= MOD;
    }
    cout << R;
}
```

BOJ 16858. 고양이 소개팅

아래에 있는 수컷 고양이부터 차례대로 아직 매칭되지 않았으면서 도달 가능한 가장 밑에 있는 암컷 고양이와 매칭하는 그리디 알고리즘이 성립합니다. 트리에서 DFS를 할 때, 현재 정점을 루트로 하는 서브트리 안에 있는 암컷 고양이들의 위치와 수를 `std::map`으로 관리하면서 `upper_bound`와 같은 메소드를 이용해 매칭하면 됩니다.

자식 정점의 `std::map`을 부모 정점으로 올릴 때 작은 집합을 큰 집합으로 합치는 small to large 테크닉을 이용하면 각 원소가 최대 $O(\log N)$번만 이동되므로 전체 시간 복잡도는 $O(N \log^2 N)$이 됩니다.

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

ll N, A[202020], V[202020], P[202020], L[202020], D[202020], R;
vector<pair<ll,ll>> G[202020];
map<ll,ll> M[202020];
```

```
void Merge(int u, int v){
    if(M[u].size() < M[v].size()) swap(M[u], M[v]);
    for(auto [a,b] : M[v]) M[u][a] += b;
}

void DFS(int v, ll d){
    for(auto [i,w] : G[v]) DFS(i, d+w);
    for(auto [i,w] : G[v]) Merge(v, i);
    if(V[v] == -1){ M[v][d] = A[v]; return; }
    auto it = M[v].upper_bound(d+V[v]);
    while(A[v] > 0 && it != M[v].begin()){
        ll now = min(A[v], (--it)->second);
        R += now; A[v] -= now; it->second -= now;
        if(!it->second) it = M[v].erase(it);
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++) cin >> V[i];
    for(int i=2; i<=N; i++) cin >> P[i];
    for(int i=2; i<=N; i++) cin >> L[i];
    for(int i=2; i<=N; i++) G[P[i]].emplace_back(i, L[i]);
    DFS(1, 0);
    cout << R;
}
```

BOJ 17675. 램프

기본적인 관찰이 몇 가지 필요합니다.

- 한 지점에 1번 연산과 2번 연산을 모두 사용할 필요 없음
- 한 지점에 3번 연산을 여러 번 사용할 필요 없음
- 3번 연산은 1, 2번 연산을 모두 끝낸 다음에 해도 됨
- 1, 2번 연산을 적용하는 구간은 서로 겹치지 않음
- 3번 연산을 적용하는 구간은 서로 겹치지 않음

조금 더 생각해 보면, 1번 연산을 사용한 구간과 2번 연산을 사용한 구간이 인접하지 않는 최적해가 항상 존재함을 알 수 있습니다. 만약 두 구간이 인접하는 최적해가 존재한다면, 두 구간의 합집합에 1번 연산을 적용한 다음 2번 연산을 사용할 구간에 3번 연산을 사용하면 되기 때문입니다.

이제, 최소 횟수로 램프를 조작하는 전략을 만들 것입니다. 1번 연산과 2번 연산을 이용해 수열에 0과 1을 적당히 배치한 다음 3번 연산을 적당히 적용할 것입니다. 1번 연산을 이용해 0으로 바꾼 위치를 0, 2번 연산을 이용해 1로 바꾼 위치를 1, 연산을 수행하지 않아서 값이 바뀌지 않은 위치를 x라고 표시합시다. 0과 1은 인접하지 않기 때문에 `xx00xx00xx11xx00xx11` 같은 꼴이 나올 것입니다.

점화식을 다음과 같이 정의합시다.

- $D(i, j) := A[1 \cdots i]$만 신경썼을 때, i번째 값에 j를 배치한 다음 $A[1 \cdots i]$와 $B[1 \cdots i]$를 동일하게 만들 때 필요한 연산의 최소 횟수
- $C(i, j) := i$번째 값에 j를 배치했을 때 $B[i]$와 동이랗면 0, 다르면 1. 즉, 3번 연산의 필요 여부를 나타내는 배열

C 배열을 이용하면 D 배열을 쉽게 계산할 수 있습니다. 상태는 총 $3N$개 있고 각 상태의 답을 $O(1)$에 구할 수 있으므로 $O(N)$ 시간에 문제를 해결할 수 있습니다.

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

int N, A[1010101], B[1010101], C[1010101][3], D[1010101][3];

int SetCost(int idx, int prv, int now){ return now != 2 && prv != now; }
int FlipCost(int idx, int prv, int now){ return C[idx-1][prv] == 0 && C[idx]
[now] == 1; }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++){ char c; cin >> c; A[i] = c - '0'; }
    for(int i=1; i<=N; i++){ char c; cin >> c; B[i] = c - '0'; }
    for(int i=1; i<=N; i++) C[i][0] = B[i] != 0, C[i][1] = B[i] != 1, C[i][2] =
A[i] != B[i];

    memset(D, 0x3f, sizeof D);
    D[1][0] = 1 + C[1][0]; D[1][1] = 1 + C[1][1]; D[1][2] = C[1][2];
    for(int i=2; i<=N; i++){
        for(int now=0; now<3; now++) for(int prv=0; prv<3; prv++) D[i][now] =
min(D[i][now], D[i-1][prv] + SetCost(i, prv, now) + FlipCost(i, prv, now));
    }
    cout << *min_element(D[N], D[N]+3);
}
```

BOJ 18664. Minimums on the Edges

i번 정점에 붙은 토큰의 개수를 A_i, 정점 U_i, V_i를 연결하는 간선의 용량을 $B_i = \min(A_{U_i}, A_{V_i})$라고 합시다. 이 문제는 아래 조건을 만족하는 A_i를 구하는 문제입니다.

- $\sum A_i \le S$: $\sum A_i < K$라면 아무 정점에 토큰을 더 붙여주면 됩니다.
- $\sum B_i$를 최대화

집합 $S_i = v | A_v \ge i$를 정의합시다. 토큰이 1개 이상 붙어있는 정점들을 덮는 덮개, 2개 이상 붙어있는 정점들을 덮는 덮개, ...라고 생각하면 편합니다. A_i는 i번 정점을 포함하는 집합(덮개)의 개수가 됩니다.

또한, B_i는 (U_i를 포함하는 집합)과 (V_i를 포함하는 집합)의 교집합의 크기 = (U_i, V_i를 모두 포함하는 교집합의 크기)가 됩니다.

$f(S_i)$를 집합 S_i에 완전히 포함되는 간선의 개수로 정의하면, $\sum B_i = \sum f(S_i)$입니다. 이제 문제의 조건을 다시 정리합시다.

1. $\sum |S_i| \le S$
2. $S_{i+1} \subset S_i$: 토큰이 $i+1$개 이상 붙은 정점은 당연히 i개 이상 붙어있습니다.
3. $\sum f(S_i)$를 최대화

2번 조건을 제외하고 1번과 3번 조건만 생각하면, 무게가 $|S_i|$이고 가치가 $f(S_i)$인 knapsack문제가 됩니다. 2^N개의 부분집합을 모두 보는 것은 $O(2^N S)$로, 너무 느립니다.

잘 생각해보면, $1 \le |S_i| \le N$이기 때문에, 크기가 i이면서 $f(S_i)$가 최대가 되는 S_i를 미리 전처리한다면 $O(NS)$에 knapsack문제를 풀 수 있습니다. 크기가 i이면서 $f(S_i)$가 최대가 되는 S_i는 $O(2^N N^2)$에 구할 수 있습니다.

이렇게 구한 최적해는 2번 조건을 항상 만족합니다.

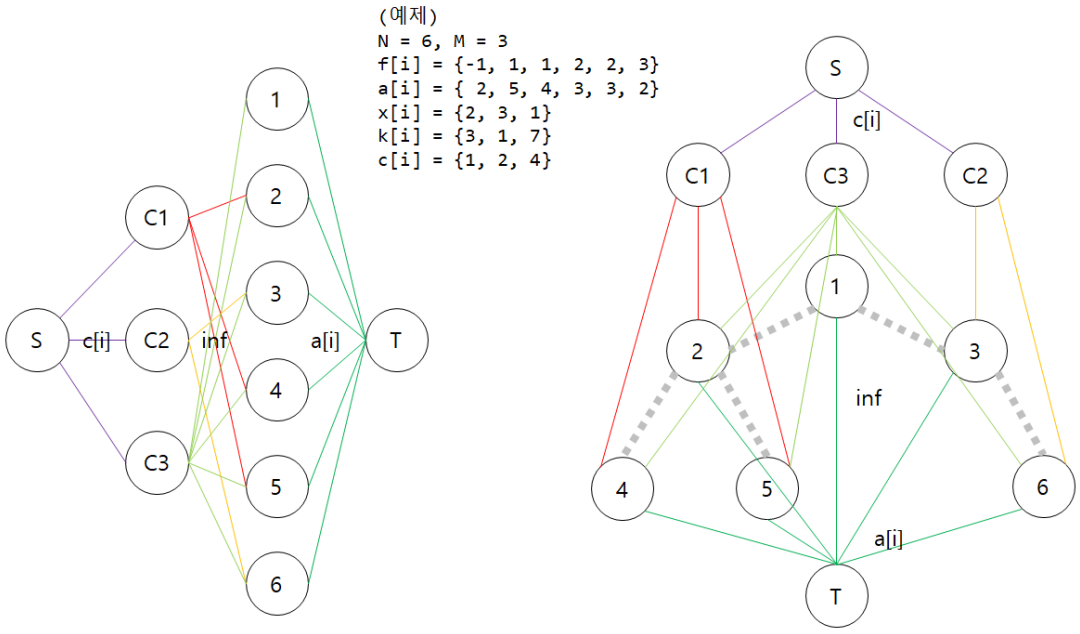
두 집합 S_i, S_j에 대해, $f(S_i) + f(S_j) \leq f(S_i \cup S_j) + f(S_i \cap S_j)$이기 때문에 두 집합이 서로 포함관계에 있는 것이 최적입니다. 그러므로 항상 2번 조건을 만족합니다.

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

int n, m, k, g[18][18], sz[1<<18], ans[18];
int fx[1<<18], c[18], val[18], dp[111], prv[111];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> n >> m >> k;
    for(int i=0; i<m; i++){
        int s, e; cin >> s >> e; s--; e--;
        g[s][e]++; g[e][s]++;
    }
    for(int i=1; i<(1<<n); i++){
        fx[i] = fx[i-(i&-i)];
        sz[i] = sz[i^(i&-i)]+1;
        int t = -1;
        for(int j=0; j<n; j++) if(i & (1 << j)) {
                if(t < 0) t = j; fx[i] += g[t][j];
            }
        if(c[sz[i]] < fx[i]) c[sz[i]] = fx[i], val[sz[i]] = i;
    }
    for(int i=1; i<=n; i++) for(int j=i; j<=k; j++) {
            if(dp[j] < dp[j-i]+c[i]) dp[j] = dp[j-i]+c[i], prv[j] = i;
        }
    int idx = 0, sum = 0;
    for(int i=0; i<=k; i++) if(dp[idx] < dp[i]) idx = i;
    for(int i=idx; i; i-=prv[i]){
        for(int j=0; j<n; j++) if(val[prv[i]] & (1 << j)) ans[j]++, sum++;
    }
    ans[0] += k-sum;
    for(int i=0; i<n; i++) cout << ans[i] << " ";
}
```

BOJ 18586. Salty Fish

(1) 카메라를 매수하고 정점을 선택하는 것과 (2) 정점을 포기하는 것, 두 가지 행동을 적절히 해서 이득을 최대화(손실을 최소화)하는 것이 문제의 목적입니다. 모든 정점을 다 먹는 것을 기본으로 하고, 각 행동을 선택함으로써 손해를 보게 되는 비용을 생각해봅시다.

카메라를 매수하는 경우 카메라의 가격인 C_i만큼 손해를 보고, 정점을 포기하는 경우 해당 정점에 있는 사과의 개수인 A_i만큼 손해를 봅니다.



Source와 카메라를 가중치가 C_i인 간선, 카메라와 그 카메라가 담당하는 정점을 가중치가 ∞인 간선, 정점과 Sink를 가중치가 A_i인 간선으로 이어주면 Min Cut문제가 됩니다. 이때 정답은 $(\sum A_i - MinCut)$이 됩니다. Max Flow = Min Cut이므로 정답은 $(\sum A_i - MaxFlow)$입니다.

그래프의 크기가 매우 크기 때문에 전형적인 플로우 알고리즘을 사용하면 안 되고, 그래프의 형태에 맞게 적절한 알고리즘을 설계해야 합니다.

기본적인 컨셉은 Ford-Fulkerson처럼 유량을 흘릴 수 있는 곳을 찾아서 흘리는 방식으로 진행합니다.

dp(v, d): v를 루트로 하는 서브트리의 정점 중, 1번 정점과 d만큼 떨어져 있는 정점에서 Sink로 가는 간선들의 잔여 유량의 합으로 정의합시다.

DFS를 하면서 아래에 있는 정점부터 처리할 것입니다.

현재 정점 v에 달려있는 카메라는 깊이가 dep[v] + k[i]인 정점까지 관리할 수 있습니다. 흘릴 수 있는 가장 깊은 곳부터 유량을 흘리는 것이 이득입니다. 그러므로 dp(v)를 std::map으로 관리하면서, prev(upper_bound)를 구해 유량을 흘려주면 됩니다.

현재 정점의 dp값을 부모 정점과 합쳐주는 것은 Small to Large를 이용하면 됩니다.

```
#include <bits/stdc++.h>
#define x first
#define y second
#define all(v) v.begin(), v.end()
#define compress(v) sort(all(v)), v.erase(unique(all(v)), v.end())
using namespace std;

typedef long long ll;
typedef pair<ll, ll> p;

ll n, m, a[303030];
vector<int> g[303030];
vector<p> ca[303030];
int par[303030], dep[303030];
map<int, ll> mp[303030];

void init(){
```

```cpp
    for(int i=1; i<=n; i++){
        g[i].clear();
        ca[i].clear();
        mp[i].clear();
        a[i] = par[i] = dep[i] = 0;
    }
}

ll flow;
void merge(map<int, ll> &mp1, map<int, ll> &mp2){
    if(mp1.size() < mp2.size()) swap(mp1, mp2);
    for(auto i : mp2) mp1[i.x] += i.y;
}
void dfs(int v){
    dep[v] = dep[par[v]] + 1;
    for(auto i : g[v]) dfs(i);

    mp[v][dep[v]] += a[v];
    for(auto i : g[v]) merge(mp[v], mp[i]);

    for(auto i : ca[v]){
        ll ff = i.y;
        while(ff && mp[v].size()){
            auto it = mp[v].upper_bound(dep[v] + i.x);
            if(it == mp[v].begin()) break; --it;
            ll now = min(ff, it->y);
            flow += now; ff -= now; it->y -= now;
            if(!it->y) mp[v].erase(it);
        }
    }
}

void solve(){
    cin >> n >> m; init();
    for(int i=2; i<=n; i++){
        cin >> par[i]; g[par[i]].push_back(i);
    }
    for(int i=1; i<=n; i++) cin >> a[i];
    for(int i=1; i<=m; i++){
        int x, k, c; cin >> x >> k >> c;
        ca[x].emplace_back(k, c);
    }
    flow = 0; dfs(1);
    cout << accumulate(a+1, a+n+1, 0ll) - flow << "\n";
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int t; cin >> t; while(t--) solve();
}
```