2023 SCCC 봄 #1

BOJ 27496. 발머의 피크 이론

i번째 시점에 마신 술은 $[i, i+L)$ 구간에서의 혈중 알코올 농도를 a_i 만큼 증가시킵니다. 따라서 이 문제는 수열 A의 구간 $[s, e)$에 어떤 값 v를 더하는 연산을 N번 수행한 다음, $129 \le A_i \le 138$를 만족하는 원소의 개수를 구하는 문제라고 생각할 수 있습니다.

구간 $[s, e)$에 어떤 값 v를 더하는 것은 $A_s, A_{s+1}, \cdots, A_N$에 v를 더하고, $A_e, A_{e+1}, \cdots, A_N$에 $-v$를 더하는 것과 같습니다. 따라서 구간에 값을 더하는 연산 (s, e, v)가 주어지면 A_s에 v, A_e에 $-v$를 더한 다음, 마지막에 A의 누적 합 배열 $S_i = \sum_{k=1}^{i} A_k$를 구하는 방식으로 연산을 효율적으로 수행할 수 있습니다.

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

int N, K, A[1010101], R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1,t; i<=N; i++) cin >> t, A[i] += t, A[i+K] -= t;
    partial_sum(A+1, A+N+1, A+1);
    for(int i=1; i<=N; i++) R += 129 <= A[i] && A[i] <= 138;
    cout << R;
}
```

BOJ 27725. 지수를 더하자

b_i의 값은 i에 p_j가 곱해진 횟수들의 합과 같습니다. 따라서 이 문제는 $1, 2, \cdots, K$에 소수 p_j가 총 몇 번 곱해졌는지 구하는 문제라고 생각할 수 있습니다.

n 이하인 자연수 중에서 d의 배수인 수의 개수는 $\lfloor n/d \rfloor$와 같습니다. 하지만 어떤 수에 소수가 여러 번 곱해져 있을 수 있기 때문에 단순히 $\lfloor K/p_j \rfloor$를 구하는 것으로는 문제를 해결할 수 없습니다.

p가 e번 곱해진 정수 n은 p와 서로소인 적당한 정수 a에 대해 $n = a \times p^e$꼴로 표현할 수 있습니다.
$n = a \times p^e$를 정확히 e번 셀 수 있는 방법을 찾아야 합니다.

$n = a \times p^e$는 $p^1, p^2, \cdots, p^{e-1}, p^e$의 배수이지만 p^{e+1}의 배수는 아닙니다. 이 점을 이용하면 모든 p_j에 대해 $\lfloor K/p_j \rfloor, \lfloor K/p_j^2 \rfloor, \lfloor K/p_j^3 \rfloor, \cdots$의 합을 구하면 된다는 것을 알 수 있습니다. $p_j^e > K$인 e는 확인할 필요가 없으므로 지수는 $\log_{p_j} K$까지만 확인해도 충분합니다.

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

ll N, K, R;
vector<ll> P;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N; P.resize(N);
    for(auto &i : P) cin >> i;
```

```
    cin >> K;
    for(auto i : P){
        ll now = i;
        while(now <= K) R += K / now, now *= i;
    }
    cout << R;
}
```

BOJ 27651. 벌레컷

입력으로 누적 합 배열이 주어진다고 가정하면, 문제에서 주어진 조건은 $A_X < A_N - A_Y < A_Y - A_X$로 생각할 수 있습니다. 이 조건을 만족하는 (X, Y)를 구해야 하는데, 고정된 Y에 대해 조건을 만족하는 X의 개수를 구하는 방식으로 접근하는 것이 편합니다.

Y가 고정되어 있을 때 선택 가능한 X의 조건을 생각해 봅시다.

- $A_X < A_N - A_Y$가 성립해야 합니다.
- $A_N - A_Y < A_Y - A_X$가 성립해야 하므로 $A_X < 2A_Y - A_N$이 성립해야 합니다.
- $A_X < A_Y - A_X$가 성립해야 하므로 $A_X < A_Y/2$가 성립해야 합니다.

따라서 우리는 Y가 고정되어 있을 때, $A_X < \min\{A_N - A_Y, 2A_Y - A_N, A_Y/2\}$를 만족하는 $1 \le X < Y$의 개수를 구해야 합니다. 문제의 입력 조건에 의해 $A_i < A_{i+1}$을 만족하므로 각 Y마다 이분 탐색을 이용해 $O(\log N)$ 시간에 X의 개수를 계산할 수 있습니다.

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

ll N, A[1010101], R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++)  cin >> A[i];
    partial_sum(A+1, A+N+1, A+1);
    for(int i=2; i<N; i++){
        ll now = min({A[N]-A[i]-1, (A[i]-1)/2, 2*A[i]-A[N]-1});
        int idx = upper_bound(A+1, A+i, now) - A;
        R += idx - 1;
    }
    cout << R;
}
```

BOJ 20445. 구간 겹치기

수직선 상의 구간 $[a, b]$를 덮는 것을 s번 정점에서 b번 정점으로 가는 것이라고 생각해 봅시다. $[s, e]$를 하나의 구간으로 덮는 것은, $s \le i \le j \le e$인 모든 i, j에 대해, i에서 j로 가는 지름길을 만드는 것이라고 생각할 수 있습니다.

입력으로는 $[-10^9, 10^9]$까지 주어질 수 있지만, 좌표 압축을 하면 정점의 개수를 $2N$개 이하로 줄일 수 있습니다.

간선은 구간마다 $O(N^2)$개씩 만들어서 총 $O(N^3)$개의 간선을 만들 수도 있고, 아니면 각 정점마다 더미 정점을 하나씩 추가해서 구간마다 $O(N)$개씩 총 $O(N^2)$개의 간선을 만들 수도 있습니다.

아무튼 $O(N)$개의 정점과 $O(N^3)$개 이하의 간선을 만들었다면 Floyd-Warshall algorithm을 이용해 모든 정점 쌍에 대한 정답을 구할 수 있습니다.

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

ll N, Q, K, V, A[111], B[111], D[333][333];
vector<ll> C;
inline void AddEdge(int s, int e, ll w){ D[s][e] = min(D[s][e], w); }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> Q; C.reserve(N+N);
    for(int i=0; i<N; i++) cin >> A[i] >> B[i];
    for(int i=0; i<N; i++) C.push_back(A[i]), C.push_back(B[i]);
    sort(C.begin(), C.end()); C.erase(unique(C.begin(), C.end()), C.end());
    for(int i=0; i<N; i++) A[i] = lower_bound(C.begin(), C.end(), A[i]) -
C.begin();
    for(int i=0; i<N; i++) B[i] = lower_bound(C.begin(), C.end(), B[i]) -
C.begin();

    K = C.size(); V = N + K;
    memset(D, 0x3f, sizeof D);
    for(int i=0; i<V; i++) AddEdge(i, i, 0);
    for(int i=0; i<N; i++) for(int j=A[i]; j<=B[i]; j++) AddEdge(j, K+i, C[B[i]]
- C[A[i]] + 1);
    for(int i=0; i<N; i++) for(int j=A[i]; j<=B[i]; j++) AddEdge(K+i, j, 0);
    for(int k=0; k<V; k++) for(int i=0; i<V; i++) for(int j=0; j<V; j++) D[i][j]
= min(D[i][j], D[i][k] + D[k][j]);

    for(int q=1; q<=Q; q++){
        int s, e; cin >> s >> e;
        if(s < C[0] || C.back() < e){ cout << "-1\n"; continue; }
        s = upper_bound(C.begin(), C.end(), s) - C.begin() - 1;
        e = lower_bound(C.begin(), C.end(), e) - C.begin();
        cout << (D[s][e] < 1e18 ? D[s][e] : -1) << "\n";
    }
}
```

BOJ 21725. 더치페이

각 사람이 지출한 금액과 실제로 지출해야 하는 금액을 어떻게 잘 계산했다고 가정하고, 송금 과정을 구하는 방법부터 알아봅시다. 한 명이 **"은행"** 역할을 해서 돈을 적게 낸 사람은 은행에게 송금하고, 돈을 많이 낸 사람은 은행에게 돈을 받으면 $n-1$번의 송금만 필요합니다.

Union-Find를 이용해 그룹이 합칠 때마다 새로운 정점을 추가하면 그룹이 합쳐지는 과정을 트리로 나타낼 수 있습니다. 그룹이 돈을 지불할 때마다 그룹을 나타내는 정점에 지불한 돈을 더하면, 각 사람이 지불해야 하는 금액은 트리에서 자신의 조상 정점의 가중치를 모두 더한 것이 됩니다. 이는 DFS를 이용해 계산할 수 있습니다.

Union-Find 과정에서 $O(N \log N)$, DFS를 이용해 지불해야 하는 금액을 계산하는데 $O(N)$, 송금 과정을 복원하는데 $O(N)$이 걸리므로 전체 시간 복잡도는 $O(N \log N)$입니다.

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

```cpp
using ll = long long;
using tl3 = tuple<ll,ll,ll>;

int P[101010], Node[101010], Size[101010];
int Find(int v){ return v == P[v] ? v : P[v] = Find(P[v]); }
void Merge(int u, int v){ Size[Find(v)] += Size[Find(u)]; P[Find(u)] = Find(v);
}
int ID(int v){ return Node[Find(v)]; }
int SZ(int v){ return Size[Find(v)]; }

int N, Q, pv;
ll A[101010], C[202020], S[202020];
vector<int> G[202020];

void Union(int u, int v){
    pv++;
    G[pv].push_back(ID(u));
    G[pv].push_back(ID(v));
    Merge(u, v); Node[Find(v)] = pv;
}
void Use(int a, ll b){
    A[a] += b; C[ID(a)] += b/SZ(a);
}
void DFS(int v){
    S[v] += C[v];
    for(auto i : G[v]) S[i] += S[v], DFS(i);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    iota(P, P+101010, 0);
    iota(Node, Node+101010, 0);
    fill(Size, Size+101010, 1);
    cin >> N >> Q; pv = N;
    for(int q=1; q<=Q; q++){
        ll op, a, b; cin >> op >> a >> b;
        if(op == 1) Union(a, b);
        else Use(a, b);
    }
    DFS(ID(1));

    vector<tl3> Res;
    for(int i=2; i<=N; i++){
        ll val = S[i] - A[i];
        if(val > 0) Res.emplace_back(i, 1, val);
        if(val < 0) Res.emplace_back(1, i, -val);
    }
    cout << Res.size() << "\n";
    for(auto [s,e,w] : Res) cout << s << " " << e << " " << w << "\n";
}
```

BOJ 11475. Journey to the “The World’s Start”

역방향으로 이동하면 항상 손해이므로 항상 번호가 증가하는 정류장만 방문한다고 가정해도 충분합니다. 지하철을 타고 이동하는 시간은 항상 $N-1$로 일정하므로 환승 시간을 최소화하는 문제라고 생각할 수 있습니다.

범위가 r인 카드는 $r-1$인 카드가 이동할 수 있는 범위를 포함합니다. 따라서 범위가 r인 카드의 가격을 $\min\{p_1, p_2, \cdots, p_r\}$이라고 생각해도 됩니다. 이동 범위가 증가하면 가격이 단조 증가하고 소요 시간은 단조 감소하므로 이동 범위에 대한 파라메트릭을 시도할 수 있습니다.

이동 범위를 R로 제한된 상황에서의 최소 소요 시간은 동적 계획법을 이용해 계산할 수 있습니다. i번째 정류장까지 이동하는데 드는 최소 환승 시간을 $D(i)$라고 하면 $D(i) = \min_{i-R\le j<i} D(j) + 1$이 성립합니다. 단순하게 계산하면 $O(N^2)$이지만 세그먼트 트리를 이용하면 $O(N \log N)$, 덱을 이용하면 $O(N)$에 계산할 수 있습니다.

결정 문제를 $O(N \log N)$ 또는 $O(N)$에 해결할 수 있으므로 전체 문제를 $O(N \log^2 N)$ 또는 $O(N \log N)$ 시간에 해결할수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int SZ = 1 << 16;
constexpr ll INF = 0x3f3f3f3f3f3f3f3f;

ll N, K, Cost[SZ], Wait[SZ], D[SZ], T[SZ<<1];
void Update(int x, ll v){
    for(T[x|=SZ]=v; x>>=1; ) T[x] = min(T[x<<1], T[x<<1|1]);
}
ll Query(int l, int r){
    l |= SZ; r |= SZ; ll res = INF;
    while(l <= r){
        if(l & 1) res = min(res, T[l++]);
        if(~r & 1) res = min(res, T[r--]);
        l >>= 1; r >>= 1;
    }
    return res;
}

bool Check(int x){
    memset(T, 0x3f, sizeof T);
    Update(1, 0);
    for(int i=2; i<=N; i++){
        int l = max(1, i-x), r = max(1, i-1);
        D[i] = Query(l, r) + Wait[i];
        Update(i, D[i]);
    }
    return D[N] + N - 1 <= K;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    for(int i=1; i<N; i++) cin >> Cost[i];
    for(int i=2; i<N; i++) cin >> Wait[i];
    int l = 1, r = N - 1;
    while(l < r){
```

```
        int m = (l + r) / 2;
        if(Check(m)) r = m;
        else l = m + 1;
    }
    cout << *min_element(Cost+r, Cost+N);
}
```

BOJ 26034. Keyboard Queries

문자열의 특정 부분 문자열 $S[l\cdots r]$이 팰린드롬이라는 정보 몇 개만 갖고 $S[a\cdots b]=S[c\cdots d]$인지 판별하는 쿼리를 처리해야 합니다. 문자열이 같은지 판별하는 것은 해싱, 문자열의 i번째와 j번째 글자가 같다는 정보는 Union-Find로 관리하는 것이 자연스러운 발상입니다.

S의 부분 문자열 $S[l\cdots r]$이 팰린드롬이라는 정보가 주어지면 $Union(l,r), Union(l+1,r-1), Union(l+2,r-2),\cdots$를 수행할 것입니다. 당연히 매번 $O(N)$번의 Union 연산을 수행하면 안 되기 때문에, 파라메트릭 서치를 이용해 "아직 합쳐지지 않은 가장 바깥 문자"를 찾아서 Union할 것입니다.

파라메트릭 서치를 수행하기 위해서는 "바깥 x문자가 모두 동일한가?", 즉 길이가 x인 접두사가 접미사를 뒤집은 것과 같은지 판별하는 결정 문제를 해결할 수 있어야 합니다. 그리고 이러한 결정 문제는 S 뒤에 S를 뒤집어서 이어붙인 문자열을 들고 있다면 2번 쿼리와 동일한 방식으로 해결할 수 있습니다.

이제 자료구조를 구체적으로 설계해 봅시다.

문자열은 S 뒤에 S를 뒤집은 것을 한 번 더 이어붙인 문자열을 관리할 것입니다. 문자열의 해시값을 세그먼트 트리로 관리할 것이고, 각 문자의 해시값은 Union-Find 상에서의 집합 번호입니다.

2번 쿼리는 단순히 세그먼트 트리에서 두 구간 $[a,b]$와 $[c,d]$에 쿼리를 날려서 결과물이 같은지 확인하면 됩니다. 아래 코드의 `EqualRange` 함수 부분입니다.

1번 쿼리는 파라메트릭 서치를 이용해 Union할 문자의 위치를 찾아서 Union하고 변경된 정보를 세그먼트 트리에 반영합니다. 파라메트릭 서치의 결정 문제에서는 접두사가 접미사를 뒤집은 것과 같은지 확인해야 하는데, 접미사를 뒤집은 것은 S를 뒤집어서 붙인 부분에서 탐색하면 2번 쿼리와 동일한 방법으로 해결할 수 있습니다. 아래 코드의 `MergeRange` 함수 부분입니다.

2번 쿼리는 세그먼트 트리에 쿼리를 2번 날리는 것이 전부이므로 쿼리당 $O(\log N)$입니다. 1번 쿼리에서 Union은 최대 $O(N)$번 호출되므로 결정 문제를 $O(N\log N)$번 호출하게 되고, 각 결정 문제는 2번 쿼리와 동일하게 $O(\log N)$ 시간에 해결할 수 있으므로 1번 쿼리에서 필요한 총 연산 횟수는 $O(N\log^2 N)$입니다.

따라서 전체 시간 복잡도는 $O(N\log^2 N+Q\log N)$입니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int SZ = 1 << 18, DIM = 2;
constexpr ll Pr[DIM] = {1299709, 1301021};
constexpr ll Md[DIM] = {1'000'000'007, 1'000'000'009};
ll Pow[SZ][DIM];

struct Hash{
    array<ll, DIM+1> a;
    Hash(){ a.fill(0); }
    Hash(ll v){ a.fill(v); a.back() = 1; }
    Hash operator + (const Hash &t) const {
```

```
        Hash res = 0;
        for(int i=0; i<DIM; i++) res.a[i] = (a[i] * Pow[t.a.back()][i] + t.a[i])
% Md[i];
        res.a.back() = a.back() + t.a.back();
        return res;
    }
    bool operator == (const Hash &t) const { return a == t.a; }
};

Hash T[SZ << 1];

void Update(int x, ll v){
    x |= SZ; T[x] = Hash(v);
    while(x >>= 1) T[x] = T[x<<1] + T[x<<1|1];
}

Hash Query(int l, int r){
    l |= SZ; r |= SZ;
    Hash lv, rv;
    while(l <= r){
        if(l & 1) lv = lv + T[l++];
        if(~r & 1) rv = T[r--] + rv;
        l >>= 1; r >>= 1;
    }
    return lv + rv;
}

int N, Q, P[SZ];
vector<int> V[SZ];

int Find(int v){ return v == P[v] ? v : P[v] = Find(P[v]); }
void Merge(int u, int v){
    u = Find(u); v = Find(v);
    if(u == v) return;
    if(V[u].size() > V[v].size()) swap(u, v);
    for(auto i : V[u]) Update(i, v), V[v].push_back(i);
    P[u] = v;
}

void Build(){
    for(int i=0; i<DIM; i++) Pow[0][i] = 1;
    for(int i=1; i<SZ; i++) for(int j=0; j<DIM; j++) Pow[i][j] = Pow[i-1][j] *
Pr[j] % Md[j];
    for(int i=1; i<=N; i++) T[i+SZ] = T[N+N-i+1+SZ] = Hash(i);
    for(int i=SZ-1; i; i--) T[i] = T[i<<1] + T[i<<1|1];
    for(int i=1; i<=N; i++) P[i] = P[N+N-i+1] = i, V[i] = {i, N+N-i+1};
}

bool EqualRange(int a, int b, int len){
    return Query(a, a+len-1) == Query(b, b+len-1);
}

void MergeRange(int a, int b, int len){
    while(len > 0 && !EqualRange(a, b, len)){
        int l = 1, r = len;
        while(l < r){
            int m = (l + r) / 2;
            if(!EqualRange(a, b, len)) r = m;
```

```
            else l = m + 1;
        }
        Merge(a+r-1, b+r-1);
        a += r; b += r; len -= r;
    }
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> Q; Build();
    for(int i=1; i<=Q; i++){
        int op; cin >> op;
        if(op == 1){
            int a, b; cin >> a >> b;
            MergeRange(a, N+N-b+1, b-a+1);
        }
        if(op == 2){
            int a, b, c, d; cin >> a >> b >> c >> d;
            if(b - a != d - c) cout << "Not equal\n";
            else if(EqualRange(a, c, b-a+1)) cout << "Equal\n";
            else cout << "Unknown\n";
        }
    }
}
```

BOJ 27400. Restore Array

배열의 누적 합 $S_i = \sum_{k=1}^{i} A_k$를 생각해 봅시다. $[l, r]$ 구간에서 k번째로 작은 값이 0일 때와 1일 때로 나눠서 볼 것입니다.

만약 $v = 0$이면 $[l, r]$ 구간에서 1의 개수가 최대 $(r - l + 1) - (k + 1) + 1$개가 됩니다. 따라서 $S_r - S_{l-1} \le r - l - k + 1$이 성립해야 합니다.

반대로 $v = 1$이면 $[l, r]$ 구간에서 1의 개수가 최소 $(r - l + 1) - k + 1$개가 됩니다. 따라서 $S_r - S_{l-1} \ge r - l - k + 2$가 성립해야 합니다.

이런 꼴의 연립 부등식은 그래프의 최단 거리로 모델링할 수 있음이 잘 알려져 있습니다. 식을 조금 변환하면 $v = 0$일 때 $S_r \le S_{l-1} + (r - l - k + 1)$, $v = 1$일 때 $S_{l-1} \le S_r - (r - l - k + 2)$를 얻을 수 있습니다.

이 부등식 말고 필요한 부등식이 더 있는데, S_i가 $S_{i-1} + 0$ 또는 $S_{i-1} + 1$이라는 것을 보장하기 위해 $S_i \le S_{i-1} + 1$과 $S_{i-1} \le S_i$가 필요합니다.

간선 $O(N + M)$개를 잘 만들었다면 벨만 포드 알고리즘을 이용해 정답을 구할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll INF = 0x3f3f3f3f3f3f3f3f;

ll N, M, D[5050];
vector<pair<int,int>> G[5050];
inline void AddEdge(int s, int e, int w){ G[s].emplace_back(e, w); }

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

```
    cin >> N >> M;
    for(int i=1; i<=N; i++) AddEdge(i-1, i, 1);
    for(int i=1; i<=N; i++) AddEdge(i, i-1, 0);
    for(int i=0; i<M; i++){
        int l, r, k, v; cin >> l >> r >> k >> v; l++; r++;
        if(v == 0) AddEdge(l-1, r, r-l+1-k);
        if(v == 1) AddEdge(r, l-1, -(r-l+1-k+1));
    }
    memset(D, 0x3f, sizeof D); D[0] = 0;
    for(int iter=0; iter<=N; iter++){
        bool change = false;
        for(int i=0; i<=N; i++) if(D[i] != INF) for(auto [j,w] : G[i]) if(D[j] >
D[i] + w) D[j] = D[i] + w, change = true;
        if(iter == N && change){ cout << -1; return 0; }
    }
    for(int i=1; i<=N; i++) cout << D[i] - D[i-1] << " ";
}
```