**Kahn’s algo**

indeg[b]++;

if(indeg[i]==0)q.push

order.push(cur)

for(int next: g[cur]){

order[next]--;

if(order[next] ==0 ){

q.push(next);

}

}

if(order.size() == n) possible

**coin\_comb 1**

int n, x;

cin >> n >> x;

vector<int>v(n);

for (auto &x : v)

cin >> x;

vector<int64\_t>comb(x + 1, 0);

comb[0] = 1;

for (int amnt = 1; amnt <= x; ++amnt) {

for (int j = 0; j < n; ++j) {

if (amnt - v[j] >= 0) {

comb[amnt] += comb[amnt - v[j]];

comb[amnt] %= MOD;

}

}

}

cout << comb[x] << '\n';

**round\_trip**

bool dfs(int ver, int p) {

Vis[ver] = true;

Par[ver] = p;

for (int child : g[ver]) {

if (child == p)continue;

if (Vis[child]) {

Start\_Cycle = child;

End\_Cycle = ver;

return true;

}

if (dfs(child, ver))

return true;

}

return false;

}

**Flight discount**

vector<vector<long long>> min\_cost(n, {INT64\_MAX, INT64\_MAX});

min\_cost[0] = {0, 0};

struct Pos {

int pos; // the current position

bool used; // whether we've used up our discount yet

long long cost; // the cost associated with this state

};

auto cmp = [&](const Pos &a, const Pos &b) { return a.cost > b.cost; };

priority\_queue<Pos, vector<Pos>, decltype(cmp)> frontier(cmp);

frontier.push({0, false, 0});

while (!frontier.empty()) {

Pos curr = frontier.top();

frontier.pop();

long long curr\_cost = min\_cost[curr.pos][curr.used];

if (curr\_cost != curr.cost) {

continue;

}

if (curr.pos == n - 1) {

break;

}

for (auto [n, nc] : neighbors[curr.pos]) {

// if we haven't used the discount yet, try using it now

if (!curr.used) {

long long new\_cost = curr\_cost + nc / 2;

if (new\_cost < min\_cost[n][true]) {

min\_cost[n][true] = new\_cost;

frontier.push(Pos{n, true, new\_cost});

}

}

// but we can always just try the normal cost route

if (curr\_cost + nc < min\_cost[n][curr.used]) {

min\_cost[n][curr.used] = curr\_cost + nc;

frontier.push(Pos{n, curr.used, curr\_cost + nc});

}

}

}

cout << min\_cost[n – 1][1];

template <typename T>

struct lazy {

vector<T>v, t, L;

size\_t size = 0;

lazy(const vector<T>&vv) {

v = vv;

size = T(vv.size());

t.assign(size << 2 , 0);

L.assign(size << 2 , 0);

if (!vv.empty())

build(1, 1, size);

}

T mrge(T a, T b , T x = 0) {

return (a + b) + x;

}

void build(T N , T s, T e) {

if (s == e) {

t[N] = v[s];

return;

}

T mid = (s + e) >> 1;

build(N << 1, s, mid);

build(N << 1 | 1, mid + 1, e);

t[N] = mrge(t[N << 1], t[N << 1 | 1]);

}

void update(T l, T r, T v) {

update(1, 1, size, l, r, v);

}

void update(T N, T s, T e, T l, T r, T x) {

if (s > r || e < l) {

return;

}

if (s >= l && e <= r) {

t[N] += (e - s + 1) \* x;

L[N] += x;

return;

}

T m = (s + e) >> 1;

update(N << 1, s, m, l, r, x);

update(N << 1 | 1, m + 1, e, l, r, x);

t[N] = mrge(t[N << 1] , t[N << 1 | 1], (e - s + 1) \* L[N]);

}

T query(T l, T r) {

return query(1, 1, size, l, r);

}

T query(T N, T s, T e, T l, T r, T carry = 0) {

if (s > r || e < l) {

return 0;

}

if (s >= l && e <= r) {

return mrge(t[N], carry \* (e - s + 1));

}

T m = (s + e) >> 1;

T q1 = query(N << 1, s, m, l, r, carry + L[N]);

T q2 = query(N << 1 | 1, m + 1, e, l, r, carry + L[N]);

return mrge(q1, q2);

}

};

**lenght\_encoding**

template<typename T, typename T\_iterable>

vector<pair<T, int>> run\_length\_encoding(const T\_iterable &items) {

vector<pair<T, int>> runs;

T previous;

int count = 0;

for (const T &item : items)

if (item == previous) {

count++;

} else {

if (count > 0)

runs.emplace\_back(previous, count);

previous = item;

count = 1;

}

if (count > 0)

runs.emplace\_back(previous, count);

return runs;

}

**coin comb 2**

vector<vector**<int>**> dp(n+1,vector**<int>**(target+1,0));

dp[0][0] = 1;

**for** (**int** i = 1; i <= n; i++) {

**for** (**int** j = 0; j <= target; j++) {

dp[i][j] = dp[i-1][j];

**int** left = j-x[i-1];

**if** (left >= 0) {

(dp[i][j] += dp[i][left]) %= mod;

}

}

}

cout << dp[n][target] << endl;