Minimum cycle in weighted graph

```
#include <bits/stdc++.h>
#define MAX_SIZE 8011
using namespace std;

const int inf = 1e9;
int M;
vector<pair<pair<int, int>, int>> g[MAX_SIZE];
int dist[MAX_SIZE], visited[MAX_SIZE];
int edg[MAX_SIZE][5];

int dijkstra(int src, int target, int bad) {
    if (g[src].empty()) return inf;
    int ret = inf;
    memset(visited, 0, sizeof(visited));
    memset(dist, 63, sizeof(dist));
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> q;
    q.push(make_pair(0, src));
    dist[src] = 0;
    while (!q.empty()) {
        auto p = q.top();
        q.pop();
        if (visited[p.second]) continue;

        if (p.first + edg[bad][4] > ret) break;
        visited[p.second] = 1;
        dist[p.second] = p.first;

        if (p.second == target) {
            ret = min(ret, p.first + edg[bad][4]);
        }

        for (auto np : g[p.second]) {
            int nv = np.first.first, w = np.first.second, id
= np.second;
```

```
            if (visited[nv] || id == bad) continue;
            if (dist[nv] > dist[p.second] + w) {
                dist[nv] = dist[p.second] + w;
                q.push(make_pair(dist[nv], nv));
            }
        }
    }

    return ret;
}

void solve() {
    int i, j, ans = inf;
    scanf("%d", &M);
    for (i = 1; i < MAX_SIZE; ++i) g[i].clear();
    vector<pair<int, int>> tmp, disc;
    for (i = 1; i <= M; ++i) {
        for (j = 0 ; j < 5; ++j) {
            scanf("%d", &edg[i][j]);
        }

        tmp.emplace_back(edg[i][0], edg[i][1]);
        tmp.emplace_back(edg[i][2], edg[i][3]);
    }

    sort(tmp.begin(), tmp.end());

    for (i = 0 ; i < (int) tmp.size(); ++i) {
        if (i == 0 || tmp[i] != tmp[i-1])
disc.push_back(tmp[i]);
    }

    for (i = 1; i <= M; ++i) {
        auto p = make_pair(edg[i][0], edg[i][1]);
        int id1 = lower_bound(disc.begin(), disc.end(), p) -
disc.begin() + 1;
        int id2 = lower_bound(disc.begin(), disc.end(),
make_pair(edg[i][2], edg[i][3])) - disc.begin() + 1;
```

```
            if (id1 == id2) {
                ans = min(ans, edg[i][4]);
            } else {
                g[id1].emplace_back(make_pair(id2, edg[i][4]),
i);
                g[id2].emplace_back(make_pair(id1, edg[i][4]),
i);
            }
        }

        for (i = 1; i <= M; ++i) {
            int id1 = lower_bound(disc.begin(), disc.end(),
make_pair(edg[i][0], edg[i][1])) - disc.begin() + 1;
            int id2 = lower_bound(disc.begin(), disc.end(),
make_pair(edg[i][2], edg[i][3])) - disc.begin() + 1;
            ans = min(ans, dijkstra(id1, id2, i));
        }

        if (ans == inf) {
            printf("0\n");
        } else {
            printf("%d\n", ans);
        }
}

int main() {
    int T;
    scanf("%d", &T);
    for (int t = 1; t <= T; ++t) {
        printf("Case #%d: ", t);
        solve();
    }
    return 0;
}
```

Solve QBF in 2-SAT form

```
#include <bits/stdc++.h>
#define MAX_SIZE 200011
using namespace std;

char s[MAX_SIZE];
vector<int> g[MAX_SIZE];
set<int> h[MAX_SIZE];
int dp[MAX_SIZE], dp2[MAX_SIZE];
int cmpid[MAX_SIZE], low[MAX_SIZE], visited[MAX_SIZE],
instack[MAX_SIZE];
set<int> univ[MAX_SIZE], exis[MAX_SIZE];
stack<int> st;
int cnt, N, M, cid = 1;

void init() {
    int i;
    cid = 1;
    cnt = 0;
    for (i = 0; i <= 2 * N + 1; ++i) {
        cmpid[i] = low[i] = visited[i] = instack[i] = 0;
        g[i].clear();
        h[i].clear();
        dp[i] = dp2[i] = 0;
        univ[i].clear();
        exis[i].clear();
    }
    while (!st.empty()) st.pop();
}

void tarjan(int v) {
    st.push(v);
    instack[v] = 1;
    visited[v] = low[v] = ++cnt;
    for (auto nv : g[v]) {
        if (!visited[nv]) {
            tarjan(nv);
            low[v] = min(low[v], low[nv]);
```

```
        } else if (instack[nv]) {
            low[v] = min(low[v], visited[nv]);
        }
    }

    if (low[v] == visited[v]) {
        while (st.top() != v) {
            instack[st.top()] = 0;
            cmpid[st.top()] = cid;
            st.pop();
        }
        cmpid[st.top()] = cid++;
        instack[st.top()] = 0;
        st.pop();
    }
}

int pos(int x) {
    return x * 2;
}

int neg(int x) {
    return x * 2 - 1;
}

pair<int, int> cnf[MAX_SIZE];

bool isexist(int l) {
    return s[abs(l)] == 'E';
}

void addconstraint(int u, int v) {
    int p1 = u, p2 = v;
    if (p1 > 0) {
        p1 = neg(p1);
    } else {
        p1 = pos(-p1);
    }
```

```
    if (p2 > 0) {
        p2 = pos(p2);
    } else {
        p2 = neg(-p2);
    }

    g[p1].push_back(p2);
}

bool dfs(int v) {
    if (visited[v]) return dp[v];
    for (auto nv : h[v]) {
        dp[v] += dfs(nv);
    }

    visited[v] = 1;
    return dp[v];
}

bool solve() {
    int i;
    scanf("%d%d", &N, &M);
    scanf("%s", s + 1);
    init();
    for (i = 1; i <= M; ++i) {
        scanf("%d%d", &cnf[i].first, &cnf[i].second);
    }

    for (i = 1; i <= M; ++i) {
        if (abs(cnf[i].first) > abs(cnf[i].second))
swap(cnf[i].first, cnf[i].second);
        if (cnf[i].first + cnf[i].second == 0) continue;
        if (!isexist(cnf[i].first)
&& !isexist(cnf[i].second)) {
            return false;
        }
```

```
        if (!isexist(cnf[i].second)) {
            addconstraint(cnf[i].first, cnf[i].first);
        } else {
            addconstraint(cnf[i].first, cnf[i].second);
            addconstraint(cnf[i].second, cnf[i].first);
        }
    }

    for (i = 1; i <= 2 * N; ++i) {
        if (!visited[i]) {
            tarjan(i);
        }
    }

    for (i = 1; i <= N; ++i) {
        if (isexist(i) && cmpid[pos(i)] == cmpid[neg(i)])
return false;
    }

    for (i = 1; i <= 2 * N; ++i) {
        for (auto v : g[i]) {
            if (cmpid[v] != cmpid[i]) {
                h[cmpid[i]].insert(cmpid[v]);
            }
        }
    }

    for (i = 1; i <= 2 * N; ++i) {
        visited[i] = 0;
        if (!isexist((i + 1) / 2)) {
            dp[cmpid[i]] += 1;
            dp2[cmpid[i]] += 1;
            univ[cmpid[i]].insert((i + 1) / 2);
        } else {
            exis[cmpid[i]].insert((i + 1) / 2);
        }
    }
    for (i = 1; i <= 2 * N; ++i) {
```

```
        if (!visited[i]) {
            dfs(i);
        }
    }

    for (i = 1; i <= 2 * N; ++i) {
        // printf("%d %d\n", i, dp[i]);
        if (dp[i] >= 2 && dp2[i] >= 1) return false;
    }

    for (i = 1; i <= 2 * N; ++i) {
        if (!univ[i].empty() && !exis[i].empty()) {
            if (*univ[i].rbegin() > *exis[i].begin()) return
false;
        }
    }
    return true;
}

int main() {
    int T;
    scanf("%d", &T);
    while (T-- > 0) {
        bool ret = solve();
        printf("%s\n", ret ? "TRUE" : "FALSE");
    }
    return 0;
}
```

MCMF linear programming

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

/*
```

```
    This problem seems ridiculous, but it is a very common
linear programming technique, another related problem is
    https://www.luogu.com.cn/problem/P3980 (in Chinese).
    Let xi = 1 be select s[i], otherwise, we select e[i]
    The initial observation is we can take all e[i] and for
each xi taken, we get s[i] - e[i]
    ms <= x1 + x2 + ... + x_(k) <= k - me
    ms <= x2 + x3 + .... + x_(k+1) <= k - me
    ...
    ms <= x_(n-k+1) + x_(n-k+2) + ... + x_n <= k - me
    we want to maximize sum(xi(s[i] - e[i]), i = 1..n)
    The idea is to use MCMF to solve this special system of
linear programming
    we rewrite the inequalities
    ms + Y[1] = x1 + x2 + ... + x_(k) = k - me -
Z[1]                ....(1)
    ms + Y[2] = x2 + x3 + .... + x_(k+1) = k - me -
Z[2]             ....(2)
    ms + Y[3] = x3 + x4 + .... + x_(k+2) = k - me -
Z[3]             ....(3)
    ms + Y[4] = x4 + x5 + .... + x_(k+3) = k - me -
Z[4]             ....(4)
    ...
    ms + Y[n-k] = x_(n-k) + ...+ x_(n-1) = k - me - Z[n-
k]          ....(n-k)
    ms + Y[n-k+1] = x_(n-k+1) + ... + x_(n) = k - me - Z[n-
k+1]             ....(n-k+1)

    we can rearrange the equations we obtain 2k equations
(aim of this step each Y[i], Z[i] and xi appears once on the
left and once on the right):
    x1 + x2 + ... + xk             = ms +
Y[1]                       ....(1)
    Y[1] + Z[1]                    = k - me -
ms                     ....(2)
    x_(k+1) + k - ms - me          = x1 + Y[2] +
Z[1]                ....(3)
```

```
    Y[2] + Z[2]                     = k - me -
ms                      ....(4)
    x_(k+2) + k - ms - me           = x2 + Y[3] +
Z[2]                ....(5)
    ...
    x_(n) + k - ms - me             = x_(n-k) + Y[n-k+1] +
Z[n-k]     ....(2n-2k+1)
    Y[n-k+1] + Z[n-k+1]             = k - me -
ms                      ....(2n-2k+2)
    k - me                          = x_(n-k+1) + ... + x_(n)
+ Z[n-k+1]   ....(2n-2k+3)
    treat each equation as a vertex and each variable as an
edge
    if xi occurs on the left in equation e1 and on the right
in equation e2, add edge e1->e2 with capacity 1 and cost -
e[i] + s[i] (since we want min-cost)
    in details, we add edge (1, 3), (1, 5) ... (1, 2k + 1)
for x1 to xk
                    add edge (3, 2k + 3), (5, 2k+5) ... (2n-
4k+1, 2n-2k+1) for x_(k+1) to x_(n-k)
                    add edge (2n-4k+3, 2n-2k+3), (2n-4k+5,
2n-2k+3) ... (2n-2k+1, 2n-2k+3) for x_(n-k+1) to x_(n)
    if Y[i] occurs on the left in equation e1 and on the
right in equation e2, add edge e1->e2 with capcity inf and
cost 0
    in details, we add edge (i, i-1)
    if Z[i] occurs on the left in equation e1 and on the
right in equation e2, add edge e1->e2 with capacity inf and
cost 0
    in details, we add edge (i, i+1)
    if a constant occurs on the left of the equation e, we
add an edge e->target with capacity constant and cost 0
    in details, we add (3, 5, 7, 9, ... 2n-2k+1, 2n-2k+3 ->
T)
    if a constant occurs on the right of the equation e, we
add an edge src->e with capacity constant and cost 0
    in details, we add (1, 2, 4, 6, 8, ... 2n-2k + 2 -> T)
    Now, "just" run MCMF and done.
```

```
*/

typedef long long ll;
const ll INF = 1e16;
const int maxn = 2111;

struct Edge {
    int from, to;
    ll cap, flow, cost;
    int id;
    Edge(int u, int v, ll c, ll f, ll w, int
id):from(u),to(v),cap(c),flow(f),cost(w), id(id)
    {}
};

struct MCMF {
    int n, m;
    int src, target;
    vector<Edge> edges;
    vector<int> G[maxn];
    int inq[maxn];
    ll d[maxn];
    int p[maxn];
    ll a[maxn];

    MCMF() {}

    void init(int n=maxn) {
        this->n = n;
        for(int i = 0; i < n; i++) G[i].clear();
        edges.clear();
    }

    void addedge(int from, int to, ll cap, ll cost, int id=-
1) {
        edges.push_back(Edge(from, to, cap, 0, cost, id));
        edges.push_back(Edge(to, from, 0, 0, -cost, -1));
        m = edges.size();
```

```
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }

    bool SPFA(int s, int t, ll &flow, ll &cost) {
        for(int i = 0; i < n; i++) d[i] = INF;
        memset(inq, 0, sizeof(inq));
        d[s] = 0; inq[s] = 1; p[s] = 0; a[s] = INF;

        queue<int> Q;
        Q.push(s);
        while(!Q.empty()) {
            int u = Q.front(); Q.pop();
            inq[u] = 0;
            for(int i = 0; i < (int) G[u].size(); i++) {
                Edge& e = edges[G[u][i]];
                if(e.cap > e.flow && d[e.to] > d[u] +
e.cost) {
                    d[e.to] = d[u] + e.cost;
                    p[e.to] = G[u][i];
                    a[e.to] = min(a[u], e.cap - e.flow);
                    if(!inq[e.to]) { Q.push(e.to); inq[e.to]
= 1; }
                }
            }
        }
        if(d[t] == INF) return false;
        flow += a[t];
        cost += d[t] * a[t];
        for(int u = t; u != s; u = edges[p[u]].from) {
                edges[p[u]].flow += a[t];
                edges[p[u]^1].flow -= a[t];
        }
        return true;
    }

    ll mincostMaxflow(ll &cost) {
        ll flow = 0;
```

```
        cost = 0;
        while(SPFA(src, target, flow, cost));
        return flow;
    }
};

MCMF mf;
int N, K, ms, me;
ll s[maxn], e[maxn];
bool sel[maxn];

void openfile() {
    freopen("delight.in", "r", stdin);
    freopen("delight.out", "w", stdout);
}

int main() {
    #ifdef SUBMIT
    openfile();
    #endif
    int i, j, k, tolequation;
    ll cost = 0, ret = 0;
    scanf("%d%d%d%d", &N, &K, &ms, &me);
    for (i = 1; i <= N; ++i) scanf("%lld", &s[i]);
    for (i = 1; i <= N; ++i) {
        scanf("%lld", &e[i]);
        ret += e[i];
    }

    mf.init();
    mf.src = maxn - 2, mf.target = maxn - 1;
    tolequation = 2 * N - 2 * K + 3;
    // deal with the constant terms
    mf.addedge(mf.src, 1, ms, 0);
    mf.addedge(tolequation, mf.target, K - me, 0);
    for (i = 3; i < tolequation; i = i + 2) {
        mf.addedge(i, mf.target, K - ms - me, 0);
    }
```

```
    for (i = 2; i <= tolequation; i = i + 2) {
        mf.addedge(mf.src, i, K - ms - me, 0);
    }

    // deal with the X[i] edges
    for (i = 1; i <= K; ++i) {
        mf.addedge(1, min(2 * i + 1, tolequation), 1, e[i] -
s[i], i);
    }

    for (i = K + 1, j = 3, k = 2 * K + 3; i <= N; ++i, j = j
+ 2, k = k + 2) {
        mf.addedge(j, min(k, tolequation), 1, e[i] - s[i],
i);
    }

    // deal with the Y[i] and Z[i] edges
    for (j = 2; j < tolequation; j = j + 2) {
        mf.addedge(j, j - 1, INF, 0);
    }

    for (j = 2; j < tolequation; j = j + 2) {
        mf.addedge(j, j + 1, INF, 0);
    }
    mf.mincostMaxflow(cost);
    printf("%lld\n", -cost + ret);
    for (auto e : mf.edges) {
        if (e.id != -1 && e.flow == e.cap) {
            sel[e.id] = true;
        }
    }

    for (i = 1; i <= N; ++i) {
        if (sel[i]) {
            printf("S");
        } else {
            printf("E");
```

```
        }
    }
    printf("\n");
    return 0;
}
```

Johnson's shortest path

```
#include <bits/stdc++.h>
#define MAX_SIZE 2511
using namespace std;
typedef long long ll;
vector<pair<int, ll>> g[MAX_SIZE];
ll dist[MAX_SIZE], h[MAX_SIZE];
int visited[MAX_SIZE];
struct edge {
    int from, to;
    ll cost;
};

vector<edge> edg;

int W, H;

int idx(int x, int y) {
    return (x - 1) * W + y;
}

ll dijkstra(int src) {
    int i;
    memset(visited, 0, sizeof(visited));
    for (i = 1; i <= W * H; ++i) dist[i] = 1e13;
    priority_queue<pair<ll, int>, vector<pair<ll, int>>,
greater<pair<ll, int>>> q;
    dist[src] = 0;
    q.push({0, src});
```

```
    while (!q.empty()) {
        auto p = q.top();
        q.pop();
        if (visited[p.second]) continue;
        visited[p.second] = 1;
        dist[p.second] = p.first;
        for (auto nv : g[p.second]) {
            if (!visited[nv.first] && dist[nv.first] >
p.first + nv.second) {
                dist[nv.first] = p.first + nv.second;
                q.push({dist[nv.first], nv.first});
            }
        }
    }
    ll ret = 0;
    for (i = 1; i <= W * H; ++i) {
        ret += dist[i];
    }

    return ret;
}

int main() {
    int i, j;
    scanf("%d%d", &W, &H);
    // case north
    for (i = 1; i <= H; ++i) {
        for (j = 1; j <= W; ++j) {
            int c;
            scanf("%d", &c);
            if (i > 1) {
                edg.push_back(edge{idx(i, j), idx(i - 1, j),
c});
            }
        }
    }

    // case west
```

```
    for (i = 1; i <= H; ++i) {
        for (j = 1; j <= W; ++j) {
            int c;
            scanf("%d", &c);
            if (j > 1) {
                edg.push_back(edge{idx(i, j), idx(i, j - 1),
c});
            }
        }
    }

    // case south
    for (i = 1; i <= H; ++i) {
        for (j = 1; j <= W; ++j) {
            int c;
            scanf("%d", &c);
            if (i < H) {
                edg.push_back(edge{idx(i, j), idx(i + 1, j),
c});
            }
        }
    }

    // case east
    for (i = 1; i <= H; ++i) {
        for (j = 1; j <= W; ++j) {
            int c;
            scanf("%d", &c);
            if (j < W) {
                edg.push_back(edge{idx(i, j), idx(i, j + 1),
c});
            }
        }
    }

    for (i = 1; i <= H; ++i) {
        for (j = 1; j <= W; ++j) {
            edg.push_back(edge{0, idx(i, j), 0});
```

```
        }
    }

    for (i = 1 ; i <= W * H; ++i) {
        h[i] = 1e13;
    }

    for (i = 1; i <= W * H; ++i) {
        for (auto e : edg) {
            if (h[e.to] > h[e.from] + e.cost) {
                h[e.to] = h[e.from] + e.cost;
            }
        }
    }

    for (auto e : edg) {
        if (e.from == 0) continue;
        g[e.from].emplace_back(e.to, e.cost - h[e.to] +
h[e.from]);
    }

    ll ans = 0, tol = (W * H) * (W * H - 1);
    for (i = 1; i <= W * H; ++i) {
        ans += dijkstra(i);
    }


    if (ans % tol != 0) {
        printf("%lld\n", ans / tol + 1);
    } else {
        printf("%lld\n", ans / tol);
    }
    return 0;
}
```