CentroidDecomposition

```
int na[MAXN], sz[MAXN], cfa[MAXN], croot;
//assume vi e[MAXN] exists

vi ce[MAXN];
vi cst;

void runCen(int);

int mib, center;

void dfssz(int now, int prv){
    sz[now] = 1;
    for(auto x: e[now])
        if(x-prv && not na[x]){
            dfssz(x, now), sz[now] += sz[x];
        }
    return;
}

void choose(int now){
    if(cst.size())
        cfa[now] = cst.back(), ce[cst.back()].pb(now);
    else
        croot = now;
    cst.pb(now);
    na[now] = 1;
    for(auto x: e[now])
        if(not na[x])
            runCen(x);
    cst.pop_back();
}

void pickCen(int now, int prv, int tot = -1){
    int mx = 0, id = -1;
    if(tot == -1) tot = sz[now];
    for(auto x: e[now])
        if(not na[x] && x-prv){
            pickCen(x, now, tot);
            if(sz[x] > mx){
                mx = sz[x];
            }
        }

    mx = max(mx, tot-sz[now]);
    if(mx < mib){
        mib = mx;
        center = now;
    }
    return;
}

void runCen(int root = 1){
    mib = INF;
    dfssz(root, root);
    pickCen(root, root);
    choose(center);
    return;
}
```

SegmentTreeAndHeavyLightDecomposition

```
// SPOJ - Can you answer these queries VII
const int MAXN = 100005, MAXL = 131072, MAXLOGN = 25; // L is the lowest
power of 2 greater than or equal to N
const int NO_MARK = 0x3fffffff;

struct Tree {
    int N;
    struct Node {
        int id, depth, size, val;
        Node *father, *hSon, *top;
        vector<Node *> neighbors;
    }nodes[MAXN];
    void addEdge(int x, int y) {
        nodes[x].neighbors.push_back(&nodes[y]);
        nodes[y].neighbors.push_back(&nodes[x]);
    }
}tree;

struct SegmentTree {
    struct Info {
```

```
        Info(): sum(0), maxSum(0), lMaxSum(0), rMaxSum(0), mark(NO_MARK) {}
        int sum, maxSum, lMaxSum, rMaxSum, mark;
        void merge(Info lSon, Info rSon) {
            // To be filled in
            sum = lSon.sum + rSon.sum;
            lMaxSum = max(lSon.lMaxSum, lSon.sum + rSon.lMaxSum);
            rMaxSum = max(rSon.rMaxSum, rSon.sum + lSon.rMaxSum);
            maxSum = max(max(lSon.maxSum, rSon.maxSum), lSon.rMaxSum +
rSon.lMaxSum);
        }
        Info reverse() {
            // To be filled in
            Info ans = *this;
            swap(ans.lMaxSum, ans.rMaxSum);
            return ans;
        }
    }nodes[MAXL << 1];
    int L;
    void buildTree() {
        L = 1;
        while (L <= tree.N) L <<= 1;
        for (int i = 1; i <= tree.N; ++i) {
            // To be filled in
            nodes[L + tree.nodes[i].id].sum = tree.nodes[i].val;
            nodes[L + tree.nodes[i].id].maxSum
            = nodes[L + tree.nodes[i].id].lMaxSum
            = nodes[L + tree.nodes[i].id].rMaxSum = max(tree.nodes[i].val,
0);
            nodes[L + tree.nodes[i].id].mark = NO_MARK;
        }
        for (int i = tree.N; i < L; ++i)
            nodes[L + i] = Info();
        for (int i = L - 1; i >= 1; --i)
            nodes[i].mark = NO_MARK, nodes[i].merge(nodes[i << 1],
nodes[(i << 1) | 1]);
    }
    void paint(int id, int l, int r, int val) { // Paint a node (for range
modification only)
        // To be filled in
        nodes[id].sum = val * (r - l);
        nodes[id].maxSum = nodes[id].lMaxSum = nodes[id].rMaxSum = max(0,
nodes[id].sum);
        nodes[id].mark = val;
    }
    void pushDown(int id, int l, int r) {
        if (id >= L) return;
        if (NO_MARK == nodes[id].mark) return;
        paint(id << 1, l, (l + r) >> 1, nodes[id].mark);
        paint((id << 1) | 1, (l + r) >> 1, r, nodes[id].mark);
        nodes[id].mark = NO_MARK;
    }
    void modify(int id, int l, int r, int p, int q, int val) {
        if (l == p && r == q) {
            paint(id, l, r, val);
            return;
        }
        int m = (l + r) >> 1;
        pushDown(id, l, r);
        if (q <= m) modify(id << 1, l, m, p, q, val);
        else if (p >= m) modify((id << 1) | 1, m, r, p, q, val);
        else modify(id << 1, l, m, p, m, val), modify((id << 1) | 1, m, r,
m, q, val);
        nodes[id].merge(nodes[id << 1], nodes[(id << 1) | 1]);
    }
    /*void modify(int x, int val) { // Single-node modification
        // To be filled in
        nodes[L + x].val += val;
        for (int i = (L + x) >> 1; i >= 1; i >>= 1)
            nodes[i].merge(nodes[i << 1], nodes[(i << 1) | 1]);
    }*/
    Info query(int id, int l, int r, int p, int q) { // nodes[id]
represents the interval [l, r); query the interval [p, q)
        if (p == l && r == q) {
            return nodes[id];
        }
        pushDown(id, l, r);
        int m = (l + r) >> 1;
        if (q <= m) return query(id << 1, l, m, p, q);
        if (p >= m) return query((id << 1) | 1, m, r, p, q);
        Info ans;
        ans.merge(query(id << 1, l, m, p, m), query((id << 1) | 1, m, r, m,
q));
        return ans;
    }
}segTree;

struct HeavyLightDecomposition {
    int cnt;
    void dfs(Tree::Node *x) { // First-round DFS; in particular, find the
heavy sons
        x->size = 1;
        int s = x->neighbors.size();
        for (int i = 0; i < s; ++i) {
            Tree::Node *y = x->neighbors[i];
            if (y != x->father) {
                y->father = x;
                y->depth = x->depth + 1;
                dfs(y);
                if (NULL == x->hSon || x->hSon->size < y->size)
                    x->hSon = y;
```

```
                x->size += y->size;
            }
        }
    }
    void dfs2(Tree::Node *x, Tree::Node *t) { // Put the tree nodes into
the segment tree
        x->id = cnt++;
        x->top = t;
        if (NULL != x->hSon) dfs2(x->hSon, t);
        int s = x->neighbors.size();
        for (int i = 0; i < s; ++i) {
            Tree::Node *y = x->neighbors[i];
            if (y != x->father && y != x->hSon) dfs2(y, y);
        }
    }
    void modify(int x, int y, int val) {
        Tree::Node *u = &tree.nodes[x], *v = &tree.nodes[y];
        Tree::Node *tu = u->top, *tv = v->top;
        while (tu != tv) {
            // Be careful whether reversions are required here
            if (tu->depth > tv->depth) {
                segTree.modify(1, 0, segTree.L, tu->id, u->id + 1, val);
                u = tu->father;
                tu = u->top;
            }
            else {
                segTree.modify(1, 0, segTree.L, tv->id, v->id + 1, val);
                v = tv->father;
                tv = v->top;
            }
        }
        if (u->depth <= v->depth) segTree.modify(1, 0, segTree.L, u->id,
v->id + 1, val);
        else segTree.modify(1, 0, segTree.L, v->id, u->id + 1, val);
    }
    int query(int x, int y) {
        SegmentTree::Info infos1[MAXLOGN], infos2[MAXLOGN], ans;
        int cnt1 = 0, cnt2 = 0;
        Tree::Node *u = &tree.nodes[x], *v = &tree.nodes[y];
        Tree::Node *tu = u->top, *tv = v->top;
        while (tu != tv) {
            if (tu->depth > tv->depth) {
                infos1[cnt1++] = segTree.query(1, 0, segTree.L, tu->id,
u->id + 1).reverse();
                u = tu->father;
                tu = u->top;
            }
            else {
                infos2[cnt2++] = segTree.query(1, 0, segTree.L, tv->id,
v->id + 1);
                v = tv->father;
                tv = v->top;
            }
        }
        if (u->depth <= v->depth)
            infos1[cnt1++] = segTree.query(1, 0, segTree.L, u->id, v->id +
1);
        else
            infos1[cnt1++] = segTree.query(1, 0, segTree.L, v->id, u->id +
1).reverse();
        for (int i = 0; i < cnt1; ++i) ans.merge(ans, infos1[i]);
        for (int i = cnt2 - 1; i >= 0; --i) ans.merge(ans, infos2[i]);
        return ans.maxSum;
    }
    void decompose() {
        tree.nodes[1].depth = 0;
        dfs(&tree.nodes[1]);
        dfs2(&tree.nodes[1], &tree.nodes[1]);
        segTree.buildTree();
    }
}hld;

void init() { // Initialize tree and hld. segTree is initialized in
SegmentTree::buildTree().
    for (int i = 1; i <= tree.N; ++i)
        tree.nodes[i].hSon = NULL, tree.nodes[i].neighbors.clear();
    hld.cnt = 0;
}

int main() {
    int Q, op, x, y, val;
    scanf("%d", &tree.N);
    init();
    for (int i = 1; i <= tree.N; ++i)
        scanf("%d", &tree.nodes[i].val);
    for (int i = 1; i < tree.N; ++i)
        scanf("%d%d", &x, &y), tree.addEdge(x, y);
    hld.decompose();
    scanf("%d", &Q);
    while (Q--) {
        scanf("%d", &op);
        if (1 == op) scanf("%d%d", &x, &y), printf("%d\n", hld.query(x,
y));
        else scanf("%d%d%d", &x, &y, &val), hld.modify(x, y, val);
    }
    return 0;
}
```

StronglyConnectedComponent_Kosaraju, 2SAT

```
const int MAXN = 100005;
int N; // Number of Vertices
```

```
vector<int> eList[MAXN], reList[MAXN]; // Edge list and reverse edge list
vector<int> vList;
bool visited[MAXN];
int sccId[MAXN];

void addEdge(int x, int y) {
    eList[x].push_back(y);
    reList[y].push_back(x);
}

void DFS(int x) {
    visited[x] = true;
    for (auto v: eList[x])
        if(not visited[v])
            DFS(v);
    vList.push_back(x);
}

void RDFS(int x, int id) {
    visited[x] = true;
    sccId[x] = id;
    for(auto v: reList[x])
        if(not visited[v])
            RDFS(v, id);
}

int findSCC() { // Returns the number of strongly connected components
    memset(visited, 0, sizeof(visited));
    vList.clear();
    for (int i = 0; i < N; ++i)
        if (!visited[i])
            DFS(i);
    memset(visited, 0, sizeof(visited));
    int nScc = 0;
    for (int i = N - 1; i >= 0; --i)
        if (!visited[vList[i]])
            RDFS(vList[i], nScc++);
    return nScc;
}

// only needed for 2SAT

#define NOTSET -1
vi cce[MAXN];
int indeg[MAXN], neg[MAXN], res[MAXN];

void propagate(int x){
    if(res[x] != NOTSET) return;

    res[x] = 0;
    res[neg[x]] = 1;

    for(auto y: cce[x])
        propagate(y);
}

bool find2SAT(){
    fill(res, res+N, NOTSET);
    int nScc = findSCC();
    fill(cce, cce+nScc, vi());
    fill(indeg, indeg+nScc, 0);
    for(int i = 0; i < N; i+=2){
        if(sccId[i] == sccId[i^1]){
            // no solution handler
            return false;
        }
    }

    queue<int> q;
    for(int i = 0, u, v; i < N; i++){
        u = sccId[i];
        for(auto j: reList[i]){
            v = sccId[j];
            if(u == v)
                continue;
            indeg[v]++;
            cce[u].pb(v);
        }
        neg[u] = sccId[i^1];
    }

    for(int i = 0; i < nScc; i++){
        if(indeg[i] == 0)
            q.push(i);
    }

    for(int i = 0; i < nScc; i++){
        if(q.empty()){
            // Severe RTE, check template
            exit(-1);
        }

        int x = q.front();

        q.pop();

        if(res[x] == NOTSET){
            // always set neg[x] to 0 or loses the point of toposort
            propagate(neg[x]);
        }
```

```
        for(auto y: cce[x]){
            if(--indeg[y] == 0)
                q.push(y);
        }
    }

    // lookup value of a[i] from res[2*i] and res[2*i+1]
    return true;
}
```

MaxFlow_Dinic

```
typedef int flow_t;
const int MAXN = 505, MAXM = 100005, DIRECTED = 0, UNDIRECTED = 1;
const flow_t INFTY = 0x3fffffff;
int N, S, T, now;
struct edge {
    flow_t remain;
    int endVertexId, nextEdgeId;
}e[MAXM << 1];
struct vertex {
    int firstEdgeId, level, firstUnsaturEdgeId;
}v[MAXN];

void _addEdge(int begin, int end, flow_t c) {
    e[now].remain = c;
    e[now].endVertexId = end;
    e[now].nextEdgeId = v[begin].firstEdgeId;
    v[begin].firstEdgeId = now++;
}

void addEdge(int begin, int end, flow_t c, int edgeType) {
    _addEdge(begin, end, c);
    _addEdge(end, begin, edgeType * c);
}

void init() {
    now = 0;
    for (int i = 0; i < N; ++i) v[i].firstEdgeId = -1;
}

bool markLevel(){
    for (int i = 0; i < N; ++i)
        v[i].level = -1, v[i].firstUnsaturEdgeId = v[i].firstEdgeId;
    v[S].level = 0;
    queue<int> Q;
    Q.push(S);
    while (!Q.empty()) {
        int x = Q.front();
        Q.pop();
        for (int i = v[x].firstEdgeId; i >= 0; i = e[i].nextEdgeId)
            if (e[i].remain && v[e[i].endVertexId].level < 0)
                v[e[i].endVertexId].level = v[x].level + 1,
Q.push(e[i].endVertexId);
    }
    return v[T].level > 0;
}

flow_t extendFlow(int x, flow_t flow) {
    if (x == T) return flow;
    flow_t t, total = 0;
    for (int &i = v[x].firstUnsaturEdgeId; i >= 0; i = e[i].nextEdgeId)
{ // Reference!
        if (v[e[i].endVertexId].level == v[x].level + 1 && e[i].remain) {
            if (t = extendFlow(e[i].endVertexId, min(flow, e[i].remain)))
                e[i].remain -= t, e[i ^ 1].remain += t, flow -= t, total
+= t;
            if (0 == flow) break;
        }
    }
    return total;
}

flow_t Dinic() {
    flow_t flow, total = 0;
    while (markLevel())
        while (flow = extendFlow(S, INFTY))
            total += flow;
    return total;
}

void buildGraph() {
    // Assign N (number of vertices), S (source) and T (sink) here.
    // Vertices are numbered from 0 to N - 1. Hence S and T should be in
[0, N).
    init();
    // Add edges here
}

int main() {
    int nCase, n, m;
    scanf("%d", &nCase);
    while (nCase--) {
        scanf("%d%d", &n, &m);
        buildGraph();
        flow_t ans = Dinic();
    }
    return 0;
}
```

MinCostFlow_Dinic

```
typedef int flow_t, cost_t;
```

```
const int MAXN = 405, MAXM = 1505, DIRECTED = 0, UNDIRECTED = 1;
const flow_t FLOW_INFTY = 0x3fffffff;
const cost_t COST_INFTY = 0x3fffffff;
int N, S, T, now, K;
bool inQ[MAXN];
struct edge {
    flow_t remain;
    cost_t cost;
    int endVertexId, nextEdgeId;
}e[MAXM << 1];
struct vertex {
    int firstEdgeId, firstUnsaturEdgeId;
    cost_t level;
}v[MAXN];

void _addEdge(int begin, int end, flow_t c, cost_t w) {
    e[now].remain = c;
    e[now].cost = w;
    e[now].endVertexId = end;
    e[now].nextEdgeId = v[begin].firstEdgeId;
    v[begin].firstEdgeId = now++;
}

void addEdge(int begin, int end, flow_t c, int edgeType, cost_t w = 1) {
    _addEdge(begin, end, c, w);
    _addEdge(end, begin, edgeType * c, -w);
}

void init() {
    now = 0;
    for (int i = 0; i < N; ++i) v[i].firstEdgeId = -1, inQ[i] = false;
}

bool markLevel(){ // SPFA
    for (int i = 0; i < N; ++i)
        v[i].level = COST_INFTY, v[i].firstUnsaturEdgeId =
v[i].firstEdgeId, inQ[i] = false;
    v[S].level = 0;
    queue<int> Q;
    Q.push(S);
    inQ[S] = true;
    while (!Q.empty()) {
        int x = Q.front();
        Q.pop();
        inQ[x] = false;
        for (int i = v[x].firstEdgeId; i >= 0; i = e[i].nextEdgeId) {
            if (e[i].remain && v[e[i].endVertexId].level > v[x].level +
e[i].cost) {
                v[e[i].endVertexId].level = v[x].level + e[i].cost;
                if (!inQ[e[i].endVertexId])
                    Q.push(e[i].endVertexId), inQ[e[i].endVertexId] =
true;
            }
        }
    }
    return v[T].level < COST_INFTY;
}

flow_t extendFlow(int x, flow_t flow) {
    if (x == T) return flow;
    inQ[x] = true;
    flow_t t, total = 0;
    for (int &i = v[x].firstUnsaturEdgeId; i >= 0; i = e[i].nextEdgeId)
{ // Reference!
        if (v[e[i].endVertexId].level == v[x].level + e[i].cost &&
e[i].remain && !inQ[e[i].endVertexId]) {
            if (t = extendFlow(e[i].endVertexId, min(flow, e[i].remain)))
                e[i].remain -= t, e[i ^ 1].remain += t, flow -= t, total
+= t;
            if (0 == flow) break;
        }
    }
    inQ[x] = false;
    return total;
}

flow_t Dinic() {
    flow_t flow, total = 0;
    cost_t cost = 0;
    while (markLevel())
        while (flow = extendFlow(S, FLOW_INFTY))
            total += flow, cost += flow * v[T].level;
    return cost; // Return total in max flow; return cost in min cost max
flow
}

void buildGraph() {
    // Assign N (number of vertices), S (source) and T (sink) here.
    // Vertices are numbered from 0 to N - 1. Hence S and T should be in
[0, N).
    init();
    // Add edges here
}
```

Splay

```
const int MAXN = 200005;

struct Splay {
    Splay *child[2], *father;
    int key, size, added;
    bool reversed;
```

```
}*root, T[MAXN];

void refresh(Splay *x) {
    x->size = 1;
    if (x->child[0] != NULL) x->size += x->child[0]->size;
    if (x->child[1] != NULL) x->size += x->child[1]->size;
    // Refresh other information here
}

void pushDown(Splay *x) {
    // Push down the labels on x
    if (x->reversed) {
        Splay *t = x->child[0];
        x->child[0] = x->child[1];
        x->child[1] = t;
        if (x->child[0] != NULL) x->child[0]->reversed ^= 1;
        if (x->child[1] != NULL) x->child[1]->reversed ^= 1;
        x->reversed = 0;
    }
    if (x->added != 0) {
        x->key += x->added;
        if (x->child[0] != NULL) x->child[0]->added += x->added;
        if (x->child[1] != NULL) x->child[1]->added += x->added;
        x->added = 0;
    }
}

void rotate(Splay *x, bool dir) {
    // x != NULL, and x->father != NULL
/*    y                       x
     / \     rotate(x, 0)     / \
    o   x   ------------->   y   o
       / \  <-------------  / \
      o   o  rotate(y, 1)  o   o   */
    Splay *y = x->father;
    pushDown(y);
    pushDown(x);
    y->child[!dir]=x->child[dir];
    if (x->child[dir] != NULL) x->child[dir]->father = y;
    x->father = y->father;
    if (y->father != NULL)
        if (y->father->child[0] == y) y->father->child[0] = x;
        else y->father->child[1] = x;
    x->child[dir] = y;
    y->father = x;
    if (y == root) root = x;
    refresh(y);
    refresh(x);
}
```

```
void splay(Splay *x, Splay *f) {
    if (x != NULL) pushDown(x);
    if (x == f || x == NULL) return;
    while (x->father != f) {
        if (x->father->father == f) {
            pushDown(x->father);
            pushDown(x);
            rotate(x, x->father->child[0] == x);
        }
        else {
            Splay *y = x->father;
            Splay *z = y->father;
            pushDown(z);
            pushDown(y);
            pushDown(x);
            if (z->child[0] == y)
                if (y->child[0] == x)
                    rotate(y, 1), rotate(x, 1);
                else
                    rotate(x, 0), rotate(x, 1);
            else
                if (y->child[0] == x)
                    rotate(x, 1), rotate(x, 0);
                else
                    rotate(y, 0), rotate(x, 0);
        }
    }
    if (f == NULL) root = x;
//  if (f != NULL) refresh(f); // Is it useful?
}

void insertAfter(Splay *x, Splay *y) { // Insert x after y
    // You should guarantee y != NULL
    splay(y, NULL); // Used to push down the labels along the path from the
root to y!
    Splay *z = y->child[1];
    if (z == NULL) {
        y->child[1] = x;
        x->father = y;
        refresh(y);
    }
    else {
        pushDown(z);
        while (z->child[0] != NULL)
            z = z->child[0], pushDown(z);
        z->child[0] = x;
        x->father = z;
        while (z != NULL)
            refresh(z), z = z->father;
    }
```

```
    splay(x, NULL);
}

Splay *selectKth(int k) { // Return the k-th element (indexing from 0)
    Splay *tree = root, *last;
    while (tree != NULL) {
        pushDown(tree);
        int leftSize = (tree->child[0] != NULL) ? tree->child[0]->size : 0;
        last = tree;
        if (leftSize == k) {
            splay(tree, NULL);
            return tree;
        }
        else if (leftSize > k) tree = tree->child[0];
        else k -= leftSize + 1, tree = tree->child[1];
    }
    splay(last, NULL);
    return NULL; // K-th element does not exist (the tree has no greater
than k elements)
}

Splay *neighbor(Splay *x, bool dir) {
    splay(x, NULL); // Used to push down the labels along the path from the
root to x!
    if (x->child[dir] == NULL) return NULL;
    x = x->child[dir];
    pushDown(x);
    while (x->child[!dir] != NULL) x = x->child[!dir], pushDown(x);
    return x;
}

Splay *prev(Splay *x) {
    return neighbor(x, 0);
}

Splay *succ(Splay *x) {
    return neighbor(x, 1);
}

void del(Splay *x) { // Delete x from the tree
    splay(x, NULL);
    if (x->child[0] == NULL) {
        root = x->child[1];
        if (x->child[1] != NULL) x->child[1]->father = NULL;
    }
    else {
        Splay *y = prev(x);
        splay(y, x);
        y->child[1] = x->child[1];
        y->father = NULL;
        if (x->child[1] != NULL) x->child[1]->father = y;
        root = y;
        refresh(y);
    }
}

int rank(Splay *x) { // Return the ranking of x (indexing from 0)
    splay(x, NULL);
    if (x->child[0] == NULL) return 0;
    return x->child[0]->size;
}

void add(int l, int r, int val) { // Add val to every element in [l, r)
    if (l > 0 && r < N) {
        Splay *x = selectKth(l - 1), *y = selectKth(r);
        splay(x, NULL);
        splay(y, x);
        if (y->child[0] != NULL)
            y->child[0]->added += val;
    }
    else if (l == 0 && r == N) {
        root->added += val;
    }
    else if (l == 0) {
        Splay *x = selectKth(r);
        splay(x, NULL);
        if (x->child[0] != NULL)
            x->child[0]->added += val;
    }
    else {
        Splay *x = selectKth(l - 1);
        splay(x, NULL);
        if (x->child[1] != NULL)
            x->child[1]->added += val;
    }
}
```

SuffixArray

```
string str;
int cnt[MAXN], RA[MAXN], tempRA[MAXN], SA[MAXN], tempSA[MAXN];

void countingSort(int k){
    int n = str.length();
    int maxi = max(SIGMA, n);
    memset(cnt, 0, sizeof(cnt));

    for(int i = 0; i < n; i++)
        cnt[i+k < n? RA[i+k]: 0]++;

    for(int i = 1; i < maxi; i++)
        cnt[i] += cnt[i-1];
```

```
    for(int i = maxi; i; i--)
        cnt[i] = cnt[i-1];
    cnt[0] = 0;

    for(int i = 0; i < n; i++)
        tempSA[cnt[SA[i]+k < n? RA[SA[i]+k]: 0]++] = SA[i];

    for(int i = 0; i < n; i++)
        SA[i] = tempSA[i];

}

void constructSA(){
    int n = str.length();
    int rank = 0;
    for(int i = 0; i < n; i++) RA[i] = str[i];
    for(int i = 0; i < n; i++) SA[i] = i;
    for(int k = 1; k < n; k<<=1){
        countingSort(k);
        countingSort(0);
        tempRA[SA[0]] = rank = 0;
        for(int i = 1; i < n; i++)
            tempRA[SA[i]] = (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] ==
RA[SA[i-1]+k]? rank: ++rank);
        for(int i = 0; i < n; i++)
            RA[i] = tempRA[i];
        if(RA[SA[n-1]] == n-1) break;
    }
}

int Phi[MAXN], PLCP[MAXN], LCP[MAXN];
void computeLCP(){
    int n = str.length();
    Phi[SA[0]] = -1;
    for(int i = 1; i < n; i++)
        Phi[SA[i]] = SA[i-1];
    for(int i = 0, len = 0; i < n; i++){
        if(Phi[i] == -1){
            PLCP[i] = 0;
            continue;
        }
        while(i+len < n && Phi[i]+len < n && str[i+len] == str[Phi[i]+len])
            len++;
        PLCP[i] = len;
        if(--len < 0) len = 0;
    }

    for(int i = 0; i < n; i++)
        LCP[i] = PLCP[SA[i]];
}

// Nanjing 2018 problem M
    constructSA();
    computeLCP();

    for(int i = RA[n+1], mi = 2e9; i < n+m+2; i++){
        mi = min(mi, LCP[i+1]);
        use[i+1] = mi;
    }
```

GaussianElimination

```
// N: number of variables; M: number of equations
const int MAXN = 505, MAXM = 1005;
const double EPS = 1e-18;
struct Gauss {
    int N, M;
    double a[MAXM][MAXN], b[MAXM];
    // Solve for ax = b, where x is a column vector
    enum {NO_SOLUTION, MANY_SOLUTION, UNIQUE_SOLUTION};
    bool doubleEq(double x, double y) {
        return x - y <= EPS && y - x <= EPS;
    }
    bool isZeroRow(int x) {
        for (int i = 1; i <= N; ++i)
            if (!doubleEq(a[x][i], 0)) return false;
        return true;
    }
    void swapRow(int x, int y) {
        for (int i = 1; i <= N; ++i)
            swap(a[x][i], a[y][i]);
        swap(b[x], b[y]);
    }
    void addRow(int x, int y, double c) { // row(y) = row(y) + c * row(x)
        for (int i = 1; i <= N; ++i)
            a[y][i] += a[x][i] * c;
        b[y] += b[x] * c;
    }
    int solve() {
        bool manySolutionFlag = false;
        int i, k = 0;
        for (i = 1; i <= N && k < N; ++i) {
            int j;
            bool flag = false;
            while (k < N && !flag) {
                ++k;
                for (j = i; j <= M; ++j) {
                    if (!doubleEq(a[j][k], 0)) {
                        if (j != i) swapRow(j, i);
                        flag = true;
                        break;
```

```
                    }
                }
                if (!flag) manySolutionFlag = true;
            }
            if (!flag) break;
            for (j = 1; j <= M; ++j)
                if (i != j) addRow(i, j, -a[j][k] / a[i][k]);
        }
        for (; i <= M; ++i)
            if (isZeroRow(i) && !doubleEq(b[i], 0)) return NO_SOLUTION;
        if (manySolutionFlag) return MANY_SOLUTION;
        for (int i = N; i >= 1; --i)
            b[i] /= a[i][i], a[i][i] = 1;
        return UNIQUE_SOLUTION;
    }
}gauss;

int main() {
    scanf("%d%d", &gauss.N, &gauss.M);
    for (int i = 1; i <= gauss.M; ++i) {
        for (int j = 1; j <= gauss.N; ++j)
            scanf("%lf", &gauss.a[i][j]);
        scanf("%lf", &gauss.b[i]);
    }
    int result = gauss.solve();
    if (Gauss::NO_SOLUTION == result) puts("No solutions");
    else if (Gauss::MANY_SOLUTION == result) puts("Many solutions");
    else {
        for (int i = 1; i <= gauss.N; ++i)
            printf("%d\n", (int)(gauss.b[i] + 0.5));
    }
    return 0;
}
```

FFT

```
typedef vector<int> VI;
double PI = acos(0) * 2;

class complex
{
public:
    double a, b;
    complex() {a = 0.0; b = 0.0;}
    complex(double na, double nb) {a = na; b = nb;}
    const complex operator+(const complex &c) const
        {return complex(a + c.a, b + c.b);}
    const complex operator-(const complex &c) const
        {return complex(a - c.a, b - c.b);}
    const complex operator*(const complex &c) const
        {return complex(a*c.a - b*c.b, a*c.b + b*c.a);}
    double magnitude() {return sqrt(a*a+b*b);}
    void print() {printf("(%.3f %.3f)\n", a, b);}
};

class FFT
{
public:
    vector<complex> data;
    vector<complex> roots;
    VI rev;
    int s, n;

    void setSize(int ns)
    {
        s = ns;
        n = (1 << s);
        int i, j;
        rev = VI(n);
        data = vector<complex> (n);
        roots = vector<complex> (n+1);
        for (i = 0; i < n; i++)
            for (j = 0; j < s; j++)
                if ((i & (1 << j)) != 0)
                    rev[i] += (1 << (s-j-1));
        roots[0] = complex(1, 0);
        complex mult = complex(cos(2*PI/n), sin(2*PI/n));
        for (i = 1; i <= n; i++)
            roots[i] = roots[i-1] * mult;
    }

    void bitReverse(vector<complex> &array)
    {
        vector<complex> temp(n);
        int i;
        for (i = 0; i < n; i++)
            temp[i] = array[rev[i]];
        for (i = 0; i < n; i++)
            array[i] = temp[i];
    }

    void transform(bool inverse = false)
    {
        bitReverse(data);
        int i, j, k;
        for (i = 1; i <= s; i++) {
            int m = (1 << i), md2 = m / 2;
            int start = 0, increment = (1 << (s-i));
            if (inverse) {
                start = n;
                increment *= -1;
            }
```

```
            complex t, u;
            for (k = 0; k < n; k += m) {
                int index = start;
                for (j = k; j < md2+k; j++) {
                    t = roots[index] * data[j+md2];
                    index += increment;
                    data[j+md2] = data[j] - t;
                    data[j] = data[j] + t;
                }
            }
        }
        if (inverse)
            for (i = 0; i < n; i++) {
                data[i].a /= n;
                data[i].b /= n;
            }
    }

    static VI convolution(VI &a, VI &b)
    {
        int alen = a.size(), blen = b.size();
        int resn = alen + blen - 1;   // size of the resulting array
        int s = 0, i;
        while ((1 << s) < resn) s++;  // n = 2^s
        int n = 1 << s;  // round up the the nearest power of two

        FFT pga, pgb;
        pga.setSize(s);  // fill and transform first array
        for (i = 0; i < alen; i++) pga.data[i] = complex(a[i], 0);
        for (i = alen; i < n; i++)    pga.data[i] = complex(0, 0);
        pga.transform();

        pgb.setSize(s);  // fill and transform second array
        for (i = 0; i < blen; i++)    pgb.data[i] = complex(b[i], 0);
        for (i = blen; i < n; i++)    pgb.data[i] = complex(0, 0);
        pgb.transform();

        for (i = 0; i < n; i++)  pga.data[i] = pga.data[i] * pgb.data[i];
        pga.transform(true); // inverse transform
        VI result = VI (resn);    // round to nearest integer
        for (i = 0; i < resn; i++)    result[i] = (int) (pga.data[i].a +
0.5);

        int actualSize = resn - 1;    // find proper size of array
        while (result[actualSize] == 0)
            actualSize--;
        if (actualSize < 0) actualSize = 0;
        result.resize(actualSize+1);
        return result;
    }
};

int main()
{
    VI a = VI (10);
    for (int i = 0; i < 10; i++)
        a[i] = (i+1)*(i+1);
    VI b = FFT::convolution(a, a);
    /* 1 8 34 104 259 560 1092 1968 3333
    5368 8052 11120 14259 17104 19234 20168 19361 16200 10000*/
    for (int i = 0; i < b.size(); i++)
        printf("%d ", b[i]);
    return 0;
}
```

NTT

```
const int MOD = (479 << 21) + 1;
const int G = 3; // Primitive root

long long fastPow(long long a, long long b) {
    long long ans = 1;
    a %= MOD;
    while (b) {
        if (b & 1) ans = (ans * a) % MOD;
        b >>= 1;
        a = (a * a) % MOD;
    }
    return ans;
}

struct NumberTheoreticTransform {
    void rearrange(long long arr[], int len) { // len must be a power of 2
        for (int i = 1, j = len >> 1; i < len - 1; ++i) {
            if (i < j) swap(arr[i], arr[j]);
            int k = len >> 1;
            while (j >= k) j -= k, k >>= 1;
            j += k;
        }
    }
    void work(long long y[], int len, int mode) {
        rearrange(y, len);
        for (int h = 2; h <= len; h <<= 1) {
            long long omegaN = fastPow(G, (MOD - 1) / h);
            if (mode == INTT) omegaN = fastPow(omegaN, MOD - 2);
            for (int j = 0, h2 = h >> 1; j < len; j += h) {
                long long omega = 1;
                for (int k = j; k < j + h2; ++k) {
                    long long a = y[k], b = (omega * y[k + h2]) % MOD;
                    y[k] = (a + b) % MOD;
                    y[k + h2] = ((a - b) % MOD + MOD) % MOD;
                    omega = (omega * omegaN) % MOD;
```

```
                }
            }
        }
        if (mode == INTT) {
            long long inv =  fastPow(len, MOD - 2);
            for (int i = 0; i < len; ++i)
                y[i] = (y[i] * inv) % MOD;
        }
    }
    enum Mode{NTT, INTT};
}ntt;

bool isRoot(long long x, long long y) { // Test if y is a primitive root of
x. Usually x is MOD, and if true is returned, we set G to y.
    long long p = y;
    for (long long i = 1; i < x - 1; ++i) {
        p = (p * y) % x;
        if (p == y) return false;
    }
    return true;
}
```

PrimeAndPhiAndMu

```
struct NumberTheory {
    static const int MAXN = 100005;
    bool isPrime[MAXN];
    int primeCount, primeList[MAXN], phi[MAXN], mu[MAXN];
    // primeCount: number of prime numbers in [1, MAXN]
    // primeList: array of all the prime numbers
    // phi: the Euler's totient function. phi[N] is the number of integers
between [1, N - 1] that are coprmime to N.
    // mu: Mobius function. mu[N] = 0 or pow(-1, number of prime factors of
N).
    // Computation of phi or mu be commented out if not needed
    NumberTheory() {
        isPrime[1] = false;
        phi[1] = 0;
        mu[1] = 1;
        for (int i = 2; i < MAXN; ++i) isPrime[i] = true;
        primeCount = 0;
        sift();
    }
    void sift() {
        for (int i = 2; i < MAXN; ++i) {
            if (isPrime[i])
                primeList[primeCount++] = i, phi[i] = i - 1, mu[i] = -1;
            for (int j = 0; j < primeCount; ++j) {
                if (i * primeList[j] >= MAXN) break;
                isPrime[i * primeList[j]] = false;
                if (i % primeList[j] == 0) {
                    phi[i * primeList[j]] = phi[i] * primeList[j];
                    mu[i * primeList[j]] = 0;
                    break;
                }
                else {
                    phi[i * primeList[j]] = phi[i] * (primeList[j] - 1);
                    mu[i * primeList[j]] = -mu[i];
                }
            }
        }
    }
}numberTheory;

int main() {
    int T, a, b, c, d, k;
    scanf("%d", &T);
    for (int nCase = 1; nCase <= T; ++nCase) {
        scanf("%d%d%d%d%d", &a, &b, &c, &d, &k);
        if (b > d) swap(b, d);
        if (k == 0) {
            printf("Case %d: 0\n", nCase);
            continue;
        }
        long long ans = 0, t = 0;
        for (int i = 1; i * k <= b; ++i)
            ans += ((long long)(b / (i * k))) * (d / (i * k)) *
numberTheory.mu[i],
            t += ((long long)(b / (i * k)) * (b / (i * k)) *
numberTheory.mu[i]);
        printf("Case %d: %I64d\n", nCase, ans - (t >> 1));
    }
    return 0;
}
```

MöbiusInversionFormula

$$F(n)=\sum_{d|n} f(d) \Rightarrow f(n)=\sum_{d|n} \mu(d) F\left(\frac{n}{d}\right)$$

$$F(n)=\sum_{n|d} f(d) \Rightarrow f(n)=\sum_{n|d} \mu\left(\frac{d}{n}\right) F(d)$$

PolicyBasedDataStructure_RBTree

```
#include <cassert>
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including
tree_order_statistics_node_update (seems unnecessary)
using namespace __gnu_pbds;

typedef tree<int, null_type, std::less<int>, rb_tree_tag,
tree_order_statistics_node_update> Set;
Set S;
```

```
int main() {
    S.insert(3); S.insert(7); S.insert(5);

    // find_by_order() returns an iterator to the k-th largest element
(counting from zero)
    assert(*S.find_by_order(2) == 7);
    assert(S.find_by_order(3) == S.end());
    assert(S.find_by_order(4) == S.end());

    // order_of_key() returns the number of items in a set that are
strictly smaller than the given item
    assert(S.order_of_key(6) == 2);
    return 0;
}
```

MOTree

```
struct query{
        int hl, hr, id, l;
        query(int l, int r, int i):hl(l), hr(r), id(i), l(0){
        }
        query(int l, int r, int i, int lca):hl(l), hr(r), id(i), l(lca){
        }
        query(){
        }
}qs[60010];

void consider(int node){
        if(st[node]) rem(node);
        else inc(node);

        st[node] ^= 1;
        return;
}

void moveTo(int hl, int hr){
        while(hl < gl)
                consider(inv[--gl]);
        while(hr > gr)
                consider(inv[++gr]);
        while(hr < gr)
                consider(inv[gr--]);
        while(hl > gl)
                consider(inv[gl++]);
        return;
}

int lca(int u, int v){
        if(d[u] < d[v]) return lca(v, u);
        for(int i = 16; i >= 0; i--)
                if(d[par[i][u]] >= d[v])
                        u = par[i][u];

        if(u == v)
                return u;

        for(int i = 16; i >= 0; i--)
                if(par[i][u] != par[i][v])
                        u = par[i][u], v = par[i][v];

        return par[0][u];
}

/* Given query range nodes {x, y}, l = lca(u, v)
        if(u == l || l == v)
                qs[i] = query(en[u], en[v], i);
        else
                qs[i] = query(ed[u], en[v], i, l);

        sort(qs, qs+m, [](query x, query y){
                if(x.hl/SQN != y.hl/SQN)
                        return x.hl/SQN < y.hl/SQN;
                if(x.hr != y.hr)
                        return x.hr < y.hr;
                return x.id < y.id;
        });

        for(int i = 0; i < m; i++){
                query q = qs[i];
                moveTo(q.hl, q.hr);
                if(q.l)  consider(q.l);
                res[q.id] = mp(q1, q2);
                if(q.l) consider(q.l);
    }*/
```

Treap

```
static unsigned long x=123456789, y=362436069, z=521288629;
unsigned long gen(void) {          //period 2^96-1
unsigned long t;
    x ^= x << 16;
    x ^= x >> 5;
    x ^= x << 1;

   t = x;
   x = y;
   y = z;
   z = t ^ x ^ y;

  return z;
}
```

```
struct node{
	int rem, res, id, pend, pendres;
	unsigned long pr;
	node *l, *r;
	node(int id, int cost):rem(cost), id(id){
		res = pend = pendres = 0;
		pr = gen(); //Use any random generating function
		l = r = nullptr;
	}
	node(){
		l = r = nullptr;
	}
};

void push(node *T){
	//push updates
	if(T->l){
		T->l->res += T->pendres;
		T->l->pendres += T->pendres;
		T->l->rem -= T->pend;
		T->l->pend += T->pend;
	}

	if(T->r){
		T->r->res += T->pendres;
		T->r->pendres += T->pendres;
		T->r->rem -= T->pend;
		T->r->pend += T->pend;
	}

	T->pendres = T->pend = 0;
	return;
}

int great(node *T){
	if(T == nullptr)
		return -INF;
	push(T);
	if(T->r == nullptr)
		return T->rem;
	return great(T->r);
}

void merge(node* &T, node *L, node *R){
	/*merging two treaps, and store it to T
		requires max(L) <= min(R)*/
	if(L == nullptr){
		T = R;
		return;
	}
	if(R == nullptr){
		T = L;
		return;
	}
	if(L->pr > R->pr){
		push(L);
		merge(L->r, L->r, R);
		T = L;
		return;
	}
	else{
		push(R);
		merge(R->l, L, R->l);
		T = R;
		return;
	}
}

void split(node *T, int val, node* &L, node* &R){
	/*
		how to split the treap:
			provide L, R placholder
			eg: R
			R->r is an auto-include
			R->l is passed as R recursively to determine
the parts which fulfills val >= v
			if there are no more candidtaes then set R->l
as nullptr
		*/
	if(T == nullptr){
		L = R = nullptr;
		return;
	}
	push(T);
	if(T->rem >= val){
		R = T;
		split(T->l, val, L, T->l);
	}
	else{
		L = T;
		split(T->r, val, T->r, R);
	}
	return;
}

void insert(node* &T, node *n){
	node *L, *R;
	split(T, n->rem, L, R);
	merge(T, L, n);
	merge(T, T, R);
```

```
        return;
}

void ext(node* &ptr, node* &ret, node* now, int val){
        /*Use split + pushall instead of this function
                having a hint pointer could greatly improve the
performance*/
        if(not now) return;
        push(now);
        if(now->rem >= val){
                ext(now->l, ret, now->l, val);
                return;
        }
        else{
                ret = now;
                merge(ptr, now->l, now->r);
                now->l = now->r = nullptr;
                return;
        }
}

void pushall(node* &L, node* &R){
        if(not R) return;
        push(R);
        pushall(L, R->l);
        pushall(L, R->r);
        insert(L, R);
        R = nullptr;
        return;
}

void unorderedMerge(node* &T, node* &L, node* &R){
        split(R, great(L), T, R);
        pushall(L, T);
        merge(T, L, R);
        return;
}
```

Wavelet

```
struct wavelet{
        vi v, ml, mr;
        int n, mi, mx;
        wavelet *lhs, *rhs;
        wavelet(){
                mi = INF, mx = -INF;
                v = ml = mr = vi();
                n = 0;
                lhs = rhs = nullptr;
        }
        void build(void){
                for(auto x: v)
                        mi = min(x, mi), mx = max(x, mx);
                if(mi == mx) return;
                int mid = mi + (1LL*mx-mi)/2;
                lhs = new wavelet();
                rhs = new wavelet();
                for(int i = 0; i < n; i++){
                        wavelet *ref = (v[i] <= mid? lhs: rhs);
                        ref->n++, ref->v.pb(v[i]);
                        ml.pb(lhs->n), mr.pb(rhs->n);
                }
                lhs->build();
                rhs->build();
                return;
        }
        int ask(int l, int r, int rank){
                if(mi == mx) return mi;
                int lt = ml[r]-(l? ml[l-1]: 0);
                if(lt >= rank)
                        return lhs->ask((l? ml[l-1]: 0), ml[r]-1,
rank);
                else
                        return rhs->ask((l? mr[l-1]: 0), mr[r]-1, rank-
lt);
        }
}
```

PersistentSegmentTree

```
struct pseg{
        static int t;
        int l, r, x;
        pseg(){
                l = r = x = 0;
        }
}T[4000010];

int pseg::t = 0, n, q, root[200010];

void upd(int &pos, int ref, int val, int l = 0+1, int r = n+1){
        pos = ++pseg::t;
        T[pos] = T[ref];
        T[pos].x++;
        if(l+1 == r)
                return;
        int mid = (l+r)/2;
        if(val < mid)
                upd(T[pos].l, T[ref].l, val, l, mid);
        else
                upd(T[pos].r, T[ref].r, val, mid, r);
}
```

```
int ask(int root, int x, int y, int l = 0+1, int r = n+1){
         if(x >= r || l >= y) return 0;
         if(x <= l && r <= y) return T[root].x;

         return ask(T[root].l, x, y, l, l+r>>1)+ask(T[root].r, x, y,
l+r>>1, r);
}
```

LCT

To query a path (u, v):
Makeroot(u), access(v), splay(v).

```
struct node{
    node *l, *r, *par;
    bool rev; int sz, oth;
    node(){
        l = r = par = NULL;
        rev = false; sz = 1, oth = 0;
    }
};

node T[200010];

void maintain(node* x){
    if(x){
        x->sz = 1+x->oth;
        if(x->l)
            x->sz += x->l->sz;
        if(x->r)
            x->sz += x->r->sz;
    }
    return;
}

bool isroot(node *x){
    if(not x->par) return true;
    return x->par->l != x && x->par->r != x;
}

void push(node* x){
    if(x && x->rev){
        if(x->l){
            x->l->rev ^= x->rev;
            swap(x->l->l, x->l->r);
        }
        if(x->r){
            x->r->rev ^= x->rev;
            swap(x->r->l, x->r->r);
        }
        x->rev = false;
    }
    return;
}

void getpushed(node* x){
    if(not isroot(x)) getpushed(x->par);
    push(x);
}

void rotate(node* x){
    node *y = x->par;
    node *z = y->par;

    //pushes can be dismissed, as they are called in splay
    push(z), push(y), push(x);
    if(not isroot(y)){
        push(z);
        if(z->l == y) z->l = x;
        if(z->r == y) z->r = x;
    }
    x->par = z;

    if(y->r == x){
        if(y->r = x->l) y->r->par = y;
        y->par = x;
        x->l = y;
    }
    else{
        if(y->l = x->r) y->l->par = y;
        y->par = x;
        x->r = y;
    }

    maintain(y), maintain(x), maintain(z);
    //ORDER IS IMPORTANT, ALWAYS UPDATE LOWER NODES FIRST
    //for z no maintain is ok as sz no change, but not always

    return;
}

void splay(node* x){
    getpushed(x);
    while(not isroot(x)){
        node *y = x->par;
        if(not isroot(y)){
            node *z = y->par;
            if((z->l == y)^(y->l == x)) rotate(x);
            else rotate(y);
```

```
        }
        rotate(x);
    }
    return;
}

void access(node* x){
    node *ret = NULL;
    while(x){
        maintain(ret);
        splay(x);
        if(x->r)
            x->oth += x->r->sz;
        x->r = ret;
        if(x->r)
            x->oth -= x->r->sz;
        ret = x;
        x = x->par;
    }
    return;
}


void makeroot(node* x){
    access(x);
    splay(x);
    x->rev ^= 1;
    swap(x->l, x->r);
    return;
}

void link(node* x, node* y){
    makeroot(x);
    makeroot(y);
    x->par = y;
    y->oth += x->sz;
    y->sz += x->sz;
    return;
}

void cut(node* x, node* y){
    makeroot(x);
    access(y);
    splay(y);
    y->l = NULL;
    x->par = NULL;
    y->sz -= x->sz;
    return;
}
```

```
bool connected(node *x, node *y){
    makeroot(x);
    access(y);
    splay(x);

    node *t = y;
    while(t->par) t = t->par;

    return t == x;
}

if(op == 1){ //link x, y
    scanf("%d%d", &x, &y);
    if(not connected(T+x, T+y)){
        link(T+x, T+y);
        deg[x]++;
        deg[y]++;
    }
    else
        printf("-1\n");
}
else if(op == 2){ //cut y's father if x, y in same cc
    scanf("%d%d", &x, &y);
    if(x != y && connected(T+x, T+y)){
        splay(T+y);
        push(T+y);
        node *fa = T[y].l;
        push(fa);
        while(fa->r){
            fa = fa->r;
            push(fa);
        }

        x = fa-T;
        cut(T+x, T+y);
        deg[x]--;
        deg[y]--;
    }
    else
        printf("-1\n");
}
```

LP Simplex

```
/ * This is a simplex solver. Given m x n matrix A, m-vector b, n-vector c,
 * finds n-vector x such that
 *  A x <= b (component-wise)
 * maximizing < x , c >
 * where <x,y> is the dot product of x and y. * /
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
```

```
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] =
A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] =
b[i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m+2; i++) if (i != r)
      for (int j = 0; j < n+2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] <
N[s]) s = j;
      }
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] <= 0) continue;
        if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
            D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r =
i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return -
numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] <
N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = {
    { 6, -1, 0 },
    { -1, -5, 0 },
    { 1, 5, 1 },
    { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -3 };
  DOUBLE _c[n] = { 1, -1, 0 };

  VVD A(m);
  VD b(_b, _b + m);
  VD c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

  LPSolver solver(A, b, c);
  VD x;
  solver.Print();
  DOUBLE value = solver.Solve(x);

  cerr << "VALUE: "<< value << endl;
  cerr << "SOLUTION:";
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
```

```
  return 0;
}
```

PalindromicTree

```
#define oddRoot (nodes)
#define evenRoot (nodes+1)
const int SIGMA = 9;
const int OFFSET = '1';

struct Node{
    Node *nxt[SIGMA], *suf;
    int len; int val;
    Node(void){
        fill(nxt, nxt+SIGMA, nullptr);
        suf = nullptr;
        len = -1;
        val = 0;
        return;
    }
    Node(Node *suf, int len): suf(suf), len(len){
        fill(nxt, nxt+SIGMA, nullptr);
        val = 0;
        return;
    }
}nodes[2000010];

class PalindromicTree{
private:
    Node* step(Node *now, char ch) const{
        while(getPrv(now->len+2) != ch || now->len+2 > str.length())
            now = now->suf;
        return now;
    }
    char getPrv(int pos) const{
        pos = str.length()-pos;
        if(pos < 0) return '\0';
        return str[pos];
    }
    void bfs(void) const{
     queue<Node*> q; q.push(oddRoot), q.push(evenRoot);
     stack<Node*> st;
     while(q.size()){
          st.push(q.front());
          q.pop();
          for(int i = 0; i < SIGMA; i++)
              if(st.top()->nxt[i])
                  q.push(st.top()->nxt[i]);
          }
          while(st.size()){
              Node *now = st.top();
              if(now->suf) now->suf->count += now->count; //push results
from long to short
                  st.pop();
          }
          return;
     }
public:
    int nodecnt;
    Node *now; string str;
    PalindromicTree(void){
        evenRoot->len = 0;
        evenRoot->suf = oddRoot;
        now = evenRoot;
        nodecnt = 2;
        return;
    }
    void insert(char ch){
        int kx = ch-OFFSET;
        str += ch;
        now = step(now, ch);
        Node *suf = step(now->suf? now->suf: now, ch);
        if(now->len == -1)
            suf = evenRoot;
        else if(suf->nxt[kx])
            suf = suf->nxt[kx];
        else{ //init, including modifying res
            nodes[nodecnt].len = now->len+2;
            nodes[nodecnt].suf = suf;
            suf = (suf->nxt[kx] = &(nodes[nodecnt++]));
        }
        if(now->nxt[kx])
            now = now->nxt[kx];
        else{ //init, including modifying res
            nodes[nodecnt].len = now->len+2;
            nodes[nodecnt].suf = suf;
            now = (now->nxt[kx] = &(nodes[nodecnt++]));
        }
        return;
    }
    void pushall(void){
     this->bfs();
     oddRoot->count = evenRoot->count = 0; //of course, remove records at
root
     }
};
```