```
// Geometria
//====POINT INCLUSION-2D===
// a Point is defined by its coordinates {int x, y;}
//==============


// isLeft(): tests if a point is Left|On|Right of an
    infinite line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2  on the line
//            <0 for P2  right of the line
//    See: Algorithm 1 "Area of Triangles and Polygons"
inline int
isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
            - (P2.x -  P0.x) * (P1.y - P0.y) );
}
//===================


// cn_PnPoly(): crossing number test for a point in a
    polygon
//      Input:   P = a point,
//               V[] = vertex points of a polygon V[n+1]
    with V[n]=V[0]
//      Return:  0 = outside, 1 = inside
// This code is patterned after [Franklin, 2000]
int
cn_PnPoly( Point P, Point* V, int n )
{
    int    cn = 0;    // the  crossing number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {    // edge from V[i]  to
        V[i+1]
        if (((V[i].y <= P.y) && (V[i+1].y > P.y))     // an
            upward crossing
            || ((V[i].y > P.y) && (V[i+1].y <=  P.y))) { //
                a downward crossing
            // compute  the actual edge-ray intersect x-
                coordinate
            float vt = (float)(P.y  - V[i].y) / (V[i+1].y -
                V[i].y);
            if (P.x <  V[i].x + vt * (V[i+1].x - V[i].x)) //
                P.x < intersect
                ++cn;   // a valid crossing of y=P.y right
                    of P.x
        }
    }
    return (cn&1);    // 0 if even (out), and 1 if  odd (in)

}
//=====================


// wn_PnPoly(): winding number test for a point in a polygon
//      Input:   P = a point,
//               V[] = vertex points of a polygon V[n+1]
    with V[n]=V[0]
//      Return:  wn = the winding number (=0 only when P is
    outside)
int
wn_PnPoly( Point P, Point* V, int n )
{
    int    wn = 0;    // the  winding number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {   // edge from V[i] to  V[i+1]
        if (V[i].y <= P.y) {          // start y <= P.y
            if (V[i+1].y  > P.y)      // an upward crossing
                if (isLeft( V[i], V[i+1], P) > 0)  // P left
                    of  edge
                    ++wn;            // have  a valid up
                        intersect
        }
        else {                        // start y > P.y (no
            test needed)
            if (V[i+1].y  <= P.y)     // a downward crossing
                if (isLeft( V[i], V[i+1], P) < 0)  // P
                    right of  edge
                    --wn;            // have  a valid down
                        intersect
        }
    }
    return wn;
}
//=========DISTANCE-POINT-PLANE-3D======
// dot product (3D) which  allows vector operations in
    arguments
#define dot(u,v)   ((u).x * (v).x + (u).y * (v).y + (u).z *
    (v).z)
#define norm(v)    sqrt(dot(v,v))  // norm = length of
    vector
#define d(P,Q)     norm(P-Q)        // distance = norm of
```

```
    difference
```

```
// dist_Point_to_Plane(): get distance (and perp base) from
    a point to a plane
//    Input:  P  = a 3D point
//            PL = a  plane with point V0 and normal n
//    Output: *B = base point on PL of perpendicular from P
//    Return: the distance from P to the plane PL
float
dist_Point_to_Plane( Point P, Plane PL, Point* B)
{
    float    sb, sn, sd;

    sn = -dot( PL.n, (P - PL.V0));
    sd = dot(PL.n, PL.n);
    sb = sn / sd;

    *B = P + sb * PL.n;
    return d(P, *B);
}

//========DISTANCE-POINT-LINE-2D=========

// dot product (3D) which allows vector operations in
    arguments
#define dot(u,v)   ((u).x * (v).x + (u).y * (v).y + (u).z *
    (v).z)
#define norm(v)     sqrt(dot(v,v))     // norm = length of
    vector
#define d(u,v)      norm(u-v)          // distance = norm of
    difference

// closest2D_Point_to_Line(): find the closest 2D Point to a
    Line
//     Input:  an array P[] of n points, and a Line L
//     Return: the index i of the Point P[i] closest to L
int
closest2D_Point_to_Line( Point P[], int n, Line L)
{
    // Get coefficients of the implicit line equation.
    // Do NOT normalize since scaling by a constant
    // is irrelevant for just comparing distances.
    float a = L.P0.y - L.P1.y;
    float b = L.P1.x - L.P0.x;
    float c = L.P0.x * L.P1.y - L.P1.x * L.P0.y;

    // initialize min index and distance to P[0]
    int mi = 0;
    float min = a * P[0].x + b * P[0].y + c;
    if (min < 0) min = -min;     // absolute value

    // loop through Point array testing for min distance to
        L
    for (i=1; i<n; i++) {
        // just use dist squared (sqrt not  needed for
            comparison)
        float dist = a * P[i].x + b * P[i].y  + c;
        if (dist < 0) dist = -dist;    // absolute value
        if (dist < min) {      // this point is closer
            mi = i;              // so have a new minimum
            min = dist;
        }
    }
    return mi;     // the index of the closest  Point P[mi]
}
//=========================
// dist_Point_to_Line(): get the distance of a point to a
    line
//     Input:  a Point P and a Line L (in any dimension)
//     Return: the shortest distance from P to L
float
dist_Point_to_Line( Point P, Line L)
{
    Vector v = L.P1 - L.P0;
    Vector w = P - L.P0;

    double c1 = dot(w,v);
    double c2 = dot(v,v);
    double b = c1 / c2;

    Point Pb = L.P0 + b * v;
    return d(P, Pb);
}
//==========================
// dist_Point_to_Segment(): get the distance of a point to a
    segment
//     Input:  a Point P and a Segment S (in any dimension)
//     Return: the shortest distance from P to S
float
dist_Point_to_Segment( Point P, Segment S)
{
    Vector v = S.P1 - S.P0;
    Vector w = P - S.P0;
```

```
    double c1 = dot(w,v);
    if ( c1 <= 0 )
        return d(P, S.P0);

    double c2 = dot(v,v);
    if ( c2 <= c1 )
        return d(P, S.P1);

    double b = c1 / c2;
    Point Pb = S.P0 + b * v;
    return d(P, Pb);
}

//=========AREA=======
// isLeft(): test if a point is Left|On|Right of an infinite
    2D line.
//    Input:  three points P0, P1, and P2
//    Return: >0 for P2 left of the line through P0 to P1
//          =0 for P2 on the line
//          <0 for P2 right of the line
inline int
isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
            - (P2.x - P0.x) * (P1.y - P0.y) );
}

// orientation2D_Triangle(): test the orientation of a 2D
    triangle
//  Input:  three vertex points V0, V1, V2
//  Return: >0 for counterclockwise
//          =0 for none (degenerate)
//          <0 for clockwise
inline int
orientation2D_Triangle( Point V0, Point V1, Point V2 )
{
    return isLeft(V0, V1, V2);
}

// area2D_Triangle(): compute the area of a 2D triangle
//  Input:  three vertex points V0, V1, V2
//  Return: the (float) area of triangle T
inline float
area2D_Triangle( Point V0, Point V1, Point V2 )
{
    return (float)isLeft(V0, V1, V2) / 2.0;
}
```

```
// orientation2D_Polygon(): test the orientation of a simple
    2D polygon
//  Input:  int n = the number of vertices in the polygon
//          Point* V = an array of n+1 vertex points with
    V[n]=V[0]
//  Return: >0 for counterclockwise
//          =0 for none (degenerate)
//          <0 for clockwise
//  Note: this algorithm is faster than computing the signed
    area.
int
orientation2D_Polygon( int n, Point* V )
{
    // first find rightmost lowest vertex of the polygon
    int rmin = 0;
    int xmin = V[0].x;
    int ymin = V[0].y;

    for (int i=1; i<n; i++) {
        if (V[i].y > ymin)
            continue;
        if (V[i].y == ymin) {   // just as low
            if (V[i].x < xmin)  // and to left
                continue;
        }
        rmin = i;      // a new rightmost lowest vertex
        xmin = V[i].x;
        ymin = V[i].y;
    }

    // test orientation at the rmin vertex
    // ccw <=> the edge leaving V[rmin] is left of the
        entering edge
    if (rmin == 0)
        return isLeft( V[n-1], V[0], V[1] );
    else
        return isLeft( V[rmin-1], V[rmin], V[rmin+1] );
}
// area2D_Polygon(): compute the area of a 2D polygon
//  Input:  int n = the number of vertices in the polygon
//          Point* V = an array of n+1 vertex points with
    V[n]=V[0]
//  Return: the (float) area of the polygon
float
area2D_Polygon( int n, Point* V )
{
    float area = 0;
    int  i, j, k;   // indices
```

```
    if (n < 3) return 0;  // a degenerate polygon

    for (i=1, j=2, k=0; i<n; i++, j++, k++) {
        area += V[i].x * (V[j].y - V[k].y);
    }
    area += V[n].x * (V[1].y - V[n-1].y);  // wrap-around term
    return area / 2.0;
}

// area3D_Polygon(): compute the area of a 3D planar polygon
//  Input:  int n = the number of vertices in the polygon
//          Point* V = an array of n+1 points in a 2D plane with V[n]=V[0]
//          Point N = a normal vector of the polygon's plane
//  Return: the (float) area of the polygon
float
area3D_Polygon( int n, Point* V, Point N )
{
    float area = 0;
    float an, ax, ay, az; // abs value of normal and its coords
    int  coord;           // coord to ignore: 1=x, 2=y, 3=z
    int  i, j, k;         // loop indices

    if (n < 3) return 0;  // a degenerate polygon

    // select largest abs coordinate to ignore for projection
    ax = (N.x>0 ? N.x : -N.x);    // abs x-coord
    ay = (N.y>0 ? N.y : -N.y);    // abs y-coord
    az = (N.z>0 ? N.z : -N.z);    // abs z-coord

    coord = 3;                    // ignore z-coord
    if (ax > ay) {
        if (ax > az) coord = 1;   // ignore x-coord
    }
    else if (ay > az) coord = 2;  // ignore y-coord

    // compute area of the 2D projection
    switch (coord) {
        case 1:
            for (i=1, j=2, k=0; i<n; i++, j++, k++)
                area += (V[i].y * (V[j].z - V[k].z));
            break;
        case 2:
            for (i=1, j=2, k=0; i<n; i++, j++, k++)
                area += (V[i].z * (V[j].x - V[k].x));
            break;
        case 3:
            for (i=1, j=2, k=0; i<n; i++, j++, k++)
                area += (V[i].x * (V[j].y - V[k].y));
            break;
    }
    switch (coord) {    // wrap-around term
        case 1:
            area += (V[n].y * (V[1].z - V[n-1].z));
            break;
        case 2:
            area += (V[n].z * (V[1].x - V[n-1].x));
            break;
        case 3:
            area += (V[n].x * (V[1].y - V[n-1].y));
            break;
    }

    // scale to get area before projection
    an = sqrt( ax*ax + ay*ay + az*az); // length of normal vector
    switch (coord) {
        case 1:
            area *= (an / (2 * N.x));
            break;
        case 2:
            area *= (an / (2 * N.y));
            break;
        case 3:
            area *= (an / (2 * N.z));
    }
    return area;
}
--------------------------
//Sort
// Merge Sort
//The merge function merges the [p1,k1) and [p2,k2) sorted ranges. The pointer k2 coincides with the end of the target
// range [p,k) meaning that no element in [p2,k2) needs to be moved if all elements from [p1,k1) are smaller than *p2.
//It is assumed that both ranges are non empty. When the range [p1,k1) becomes exhausted the merging is done because all remaining elements from [p2,k2) are already in place.
```

```cpp
template < class T >
inline  void merge (T * p1, T * k1, T * p2, T *k2)
{ T* p=p2 - (k1-p1);
    while(true)
    {if(*p1<=*p2)
    {*p++=*p1++;
        if(p1==k1) return;
    }
    else
    {*p++=*p2++;
        if(p2==k2) break;
    }
    } do
        *p++=*p1++;
    while(p1!=k1);
}

// Recursive function
template < class T >
inline void copying_mergesort (T * p, T * k, T * t)
{
    if (k > p + 16) {
        T *s = p + ((k - p) >>1);
        copying_mergesort (s, k, t+(s-p));
        copying_mergesort (p, s, s);
        merge (s,s+(s-p),t+(s-p),t+(k-p));
    } else
        copying_insertionsort (p, k, t);
}

--------------------------
// Primality
/*
 * C++ Program to Implement Fermat Primality Test
 *inretations: 50
 */
#include <cstring>
#include <iostream>
#include <cstdlib>
#define ll long long
using namespace std;
/*
 * modular exponentiation
 */
ll modulo(ll base, ll exponent, ll mod)
{
    ll x = 1;
    ll y = base;
    while (exponent > 0)
    {
        if (exponent % 2 == 1)
            x = (x * y) % mod;
        y = (y * y) % mod;
        exponent = exponent / 2;
    }
    return x % mod;
}

/*
 * Fermat's test for checking primality
 */
bool Fermat(ll p, int iterations)
{
    if (p == 1)
    {
        return false;
    }
    for (int i = 0; i < iterations; i++)
    {
        ll a = rand() % (p - 1) + 1;
        if (modulo(a, p - 1, p) != 1)
        {
            return false;
        }
    }
    return true;
}
--------------------------
//Estruturas

//AVL Tree
struct Node {
    Node *l, *r;  int h, size, key;
    Node(int k) : l(0), r(0), h(1), size(1), key(k) {}
    void u() { h=1+max(l?l->h:0, r?r->h:0);
        size=(l?l->size:0)+1+(r?r->size:0);
    }
};
Node *rotl(Node *x) { Node *y=x->r; x->r=y->l; y->l=x; x->u
    (); y->u(); return y; }
Node *rotr(Node *x) { Node *y=x->l; x->l=y->r; y->r=x; x->u
    (); y->u(); return y; }
Node *rebalance(Node *x) {
    x->u();
    if (x->l->h > 1 + x->r->h) {
        if (x->l->l->h < x->l->r->h) x->l = rotl(x->l);
```

```
        x = rotr(x);
    } else if (x->r->h > 1 + x->l->h) {
        if (x->r->r->h < x->r->l->h) x->r = rotr(x->r);
        x = rotl(x); }
    return x;
}
}

Node *insert(Node *x, int key) {
    if (x == NULL) return new Node(key);
    if (key < x->key) x->l = insert(x->l, key); else x->r =
        insert(x->r, key);
    return rebalance(x);
}

//Treap
struct Node {
    int key, aux, size;  Node *l, *r;    // BST w.r.t. key;
        min-heap w.r.t. aux
    Node(int k) : key(k), aux(rand()), size(1), l(0), r(0)
        {}
};
Node *upd(Node *p) { if(p) p->size=1+(p->l?p->l->size:0)+(p-
    >r?p->r->size:0); return p; }
void split(Node *p, Node *by, Node **L, Node **R) {
    if (p == NULL) { *L = *R = NULL; }
    else if (p->key < by->key) { split(p->r, by, &p->r, R);
        *L = upd(p); }
    else { split(p->l, by, L, &p->l); *R = upd(p); }
}
Node *merge(Node *L, Node *R) {
    Node *p;
    if (L == NULL || R == NULL) p = (L != NULL ? L : R);
    else if (L->aux < R->aux) { L->r = merge(L->r, R); p = L
        ; }
    else { R->l = merge(L, R->l); p = R; }
    return upd(p);
}
Node *insert(Node *p, Node *n) {
    if (p == NULL) return upd(n);
    if (n->aux <= p->aux) { split(p, n, &n->l, &n->r);
        return upd(n); }
    if (n->key < p->key) p->l = insert(p->l, n); else p->r =
        insert(p->r, n);
    return upd(p);
}
Node *erase(Node *p, int key) {
    if (p == NULL) return NULL;
    if (key == p->key) { Node *q = merge(p->l, p->r); delete
        p; return upd(q); }
    if (key < p->key) p->l = erase(p->l, key); else p->r =
        erase(p->r, key);
    return upd(p);
}

-------------------------- MISC
// SIMPLEX

// Two-phase simplex algorithm for solving linear programs
    of the form
// INPUT : A -- an m x n matrix
//     b -- an m-dimensional vector
//     c -- an n-dimensional vector
//     x -- a vector where the optimal solution will be
    stored
// OUTPUT: value of the optimal solution (infinity if
    unbounded above, nan if infeasible)
// To use this code, create an LPSolver object with A, b,
    and c as arguments. THen, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;
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(inti=0;i<m;i++) for (int j = 0; j < n; j++) D[i]
            [j] = A[i][j];
        for(inti=0;i<m;i++) {B[i]=n+i;D[i][n]=-1;D[i][n+1]=b
            [i];}
        for(intj=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) {
    double inv = 1.0 / D[r][s]; for(inti=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] * inv;
    for (intj=0;j <n+2;j++)if(j!=s)D[r][j]*=inv; for
        (inti=0;i <m+2;i++)if(i!=r)D[i][s]*=-inv; D[r][s
        ] = inv;
    swap(B[r], N[s]);
}
bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m; while (true) {
        ints=-1; for(intj=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; intr=-1; for(inti=0;i<m;i++){
                if (D[i][s] < EPS) 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) { intr=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,-5};
    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;
    DOUBLE value = solver.Solve(x);

    cerr << "VALUE: " << value << endl; // VALUE: 1.29032
    cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
    for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i
        ];
    cerr << endl;
    return 0;
}
```