# Inverse modulo

int gcd(ll a, ll b, ll &x, ll &y) {

    if (b==0){

        x = 1; y = 0; return a;

    }

    ll xt, yt;

    int res = gcd(b,a%b,xt,yt);

    x = yt;

    y = xt - (a/b)\*yt;

    return res;

}

int inverse(ll a, ll m){

    ll x, y;

    int g = gcd(a,m,x,y);

    if (g != 1) return -1;

    return (x%m+m)%m;

}

# Fast pow

ll powermod(int a, int b, int m){

    return b?powermod(a\*a%m,b/2,m)\*(b%2?a:1):1;

}

# Dijkstra

From 1 source to all, undirected/directed, **non-neg**

void dijkstra(int source) {

    dist.assign(n+1, INF);

    trace.assign(n+1, -1);

    priority\_queue<pi,vector<pi>,greater<pi>> q;

    q.push({0, source});

    dist[source] = 0;

    while (!q.size()) {

        int node = q.top().se;

        q.pop();

        for (int next:edges[node]) {

            int cost = dist[node]+weight[node][next];

            if (dist[next]>cost) {

                dist[next] = cost;

                trace[next] = node;

                q.push({dist[next], next});

}}}}

# Ford Bellman

From 1 source to all, undirected/directed, **neg (no neg cyc)**

void bellman(int source) {

    dist.assign(n+1, INF);

    trace.assign(n+1, -1);

    queue<int> q;

    vector<bool> inQueue(n+1, false);

    dist[source] = 0;

    q.push(source);

    inQueue[source] = true;

    while (!q.empty()){

        int node = q.front();

        q.pop();

        inQueue[node] = false;

        for (int next: edges[node]) {

            int cost = dist[node]+weight[node][next];

            if (dist[next] > cost) {

                dist[next] = cost;

                trace[next] = node;

                if (!inQueue[next]) {

                    inQueue[next] = true;

                    q.push(next);

}}}}}

# Floyd Warshall

From any pair nodes, undirected/directed, **neg (no neg cyc)**

void floyd() {

    for (int node=1;node<=n;node++) {

        for (int next: edges[node]) {

            dist[node][next] = 1;

            trace[node][next] = next;

        }

    }

    for (int node=1;node<=n;node++) {

        dist[node][node]=0;

        trace[node][node]=node;

    }

    for (int k=1;k<=n;k++){

        for (int i=1;i<=n;i++){

            for (int j=1;j<=n;j++){

                if (dist[i][j]>dist[i][k]+dist[k][j]) {

                    dist[i][j]=dist[i][k]+dist[k][j];

                    trace[i][j]=trace[i][k];

}}}}}

# DSU

struct Dsu {

    vector<int> par;

    void init(int n) {

        par.assign(n+1, 0);

        for (int i = 1; i <= n; i++) par[i] = i;

    }

    int find(int u) {

        if (par[u] == u) return u;

        return par[u] = find(par[u]);

    }

    bool join(int u, int v) {

        u = find(u); v = find(v);

        if (u == v) return false;

        par[v] = u; return true;

    }

};

# Kurskal

int kruskal(vector<Edges> edges) {

    sort(edges.begin(), edges.end(), [](Edge & x, Edge & y) {

        return x.c < y.c;

    });

    for (auto e : edges) {

        if (!dsu.join(e.u, e.v)) continue;

        totalWeight += e.c;

    }

    return totalWeight;

}

# BFS

void bfs(int source) {

    queue<int> q;

    q.push(source);

    visit[source] = true;

    while(!q.empty()){

        int node = q.front();

        q.pop();

        for (int next: edges[node]){

            if (!visit[next]){

                trace[next] = node;

                visit[next] = true;

                if (next==t) return;

                q.push(next);

}}}}

# Max-flow

void findAug(int source){

    trace.assign(n+1, -1);

    visit.assign(n+1, false);

    bfs(source);

}

void increaseFlow(int source, int terminal) {

    int u = terminal;

    int minCapacity = INF;

    while (u!=source) {

        int prev = trace[u];

        minCapacity = min(minCapacity, c[prev][u]-f[prev][u]);

        u = prev;

    }

    u = terminal;

    while (u!=source) {

        int prev = trace[u];

        f[prev][u] += minCapacity;

        f[u][prev] -= minCapacity;

        // Create new edges if needed

        u = prev;

    }

    maxFlow += minCapacity;

}

int maxFlow() {

    do {

        findAug(source);

        if (trace[terminal]!=-1){

            increaseFlow(source, terminal);

        }

    } while (trace[terminal]!=-1);

}

# Min-cost max-flow

Use Bellman instead of BFS

# Geometry

struct Point { double x; double y; };

struct Point { double x; double y;

    Point() { x = 0; y = 0; }

    Point(double iX, double iY) {

        x = iX; y = iY;

    }

};

struct Vector { double x; double y;

    Vector() { x = 0; y = 0; }

    Vector(double iX, double iY) {

        x = iX; y = iY;

    }

    Vector(Point pA, Point pB) {

        x = pB.x - pA.x;

        y = pB.y - pA.y;

    }

};

struct Line { double a, b, c;

    Line() { a = b = c = 0; };

    Line(Point pA, Point pB) {

        a = pA.y - pB.y;

        b = pB.x - pA.x;

        c = pA.x \* pB.y - pA.y \* pB.x;

    };

};

# Point distance

double distance(Vector vec) {

    return sqrt(vec.x \* vec.x + vec.y \* vec.y);

}

# Line intersection

int getIntersect(Line line1, Line line2,

double& x, double& y) {

    double D, Dx, Dy;

    D = line1.a \* line2.b - line1.b \* line2.a;

    Dx = line1.b \* line2.c - line1.c \* line2.b;

    Dy = line1.c \* line2.a - line1.a \* line2.c;

    if (abs(D) < e) {

        if (abs(Dx) < e && abs(Dy) < e) { return COINCIDED; }

        return PARALLEL;

    }

    x = Dx / D; y = Dy / D;

    return INTERSECT;

}

# Dot production

double getDot(Vector a, Vector b) {

    return a.x \* b.x + a.y \* b.y;

}

# Cosine of Point, Vector

double getCos(Point p1, Point p2, Point p3) {

    double d12, d13, d23;

    d12 = distance(p1, p2);

    d13 = distance(p1, p3);

    d23 = distance(p2, p3);

    double numeral = d12 \* d12 + d23 \* d23 - d13 \* d13;

    double denum = 2 \* d12 \* d23;

    double result = numeral / denum;

    return result;

}

double getCos(Vector a, Vector b) {

    double numeral = getDot(a, b);

    double denum = distance(a) \* distance(b);

    double result = numeral / denum;

    return result;

}

# Rotation

int CCW(Vector a, Vector b) {

    double cz = a.x \* b.y - a.y \* b.x;

    if (abs(cz) < e) { return UNCHANGE; }

    if (cz > 0) { return CCW; }

    return CC;

}

# Triangle area

double areaTriangleHeron(Point A, Point B, Point C) {

    double dAB, dBC, dAC, p;

    dAB = distance(A, B);

    dBC = distance(B, C);

    dAC = distance(A, C);

    p = (dAB + dBC + dAC) / 2;

    return sqrt(p \* (p - dAB) \* (p - dBC) \* (p - dAC));

}

double areaTriangleCross(Point A, Point B, Point C) {

    return 0.5 \* abs((B.x - A.x) \* (C.y - A.y) - (C.x - A.x) \* (B.y - A.y)); }

# Polygon

struct Poly {

    int n = 0;

    vector<Point> a;

    Poly() { n = 0; }

    Poly(int iN) { n = iN; a.assign(n, Point()); }

};

# Convex polygon area

double areaConvexPolygon(Poly p) {

    double area = 0;

    Point P0 = p.a[0];

    for (int i = 1; i < p.n - 1; i++) {

        area += areaTriangleCross(P0, p.a[i], p.a[i + 1]);

    }

    return area;

}

# Convex Hull

Poly getConvexHullWrap(vector<Point> a) {

    //Init convex set of points

    Poly result;

    int nPoint = a.size();

    //Find starting point P zero

    int indexP0 = 0;

    for (int i = 1; i < nPoint; i++) {

        if (a[indexP0].y > a[i].y ||

            (a[indexP0].y == a[i].y && a[indexP0].x > a[i].x)) {

            indexP0 = i;

        }

    }

    //Starting variable

    Vector u(-1,0);

    int indexP;

    indexP = indexP0;

    //Run until P is P0

    do {

        double maxCos = -INF;

        int indexQ = -1;

        //Each point not current P find maximum cos

        for (int i = 0; i < nPoint; i++) {

            if (i != indexP) {

                //Calculate cos value of PQ and u

                double dCos = cos(u, Vector(a[indexP], a[i]));

                if (maxCos < dCos) {

                    maxCos = dCos;

                    indexQ = i;

                }

            }

        }

        result.add(a[indexP]);

        //Assign new vector u and P

        u = Vector(a[indexP], a[indexQ]);

        indexP = indexQ;

    } while (indexP != indexP0);

    return result;

}

# Lazy segment tree

void down(int id) {

    int t = lazy[id];

    lazy[2\*id] += t;

    lazy[2\*id+1] += t;

    tree[2\*id] += t;

    tree[2\*id+1] += t;

    lazy[id] = 0;

}

void update(int id, int l, int r, int u, int v, int k) {

    if (u>r || v<l) return;

    if (u<=l && r<=v) {

        tree[id]+=k; lazy[id]+=k;

        return;

    }

    down(id);

    int mid = (l+r)/2;

    update(id\*2,l,mid,u,v,k);

    update(id\*2+1,mid+1,r,u,v,k);

    tree[id] = max(tree[id\*2],tree[id\*2+1]);

}

int get(int id, int l, int r, int u, int v) {

    if (u>r || v<l) return -INF;

    if (u<=l && r<=v) return tree[id];

    down(id);

    int mid = (l+r)/2;

    int childA = get(id\*2,l,mid,u,v);

    int childB = get(id\*2+1,mid+1,r,u,v);

    return max(childA,childB);

}

# Eratosthenes

void sieve(int n) {

    vector<bool> isPrime(n+1, true);

    isPrime[0] = isPrime[1] = false;

    for(int i = 2; i \* i <= n; i++) {

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

            isPrime[j] = false;

}}}

# LIS with BSearch

int LIS(vector<int> a){

    vector<int> b(n, INF);

    b[0] = -INF;

    int result = 0;

    for (int x: a) {

        int k = lower\_bound(b.begin(), b.end(), x) - b.begin();

        b[k] = x; result = max(result, k);

    }

}

# LCS

string LCS(string s1, string s2) {

  vector<vi> dp(m+1, vector<int>(n+1,0));

  int m = s1.length();

  int n = s2.length();

  for (int i = 1; i <= m; i++) {

    for (int j = 1; j <= n; j++) {

      if (s1[i - 1] == s2[j - 1]){

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

      } else {

        dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

      }}}

  string res = '';

  int i = m, j = n;

  while (i > 0 && j > 0) {

    if (s1[i-1]==s2[j-1]) {

      res = s1[i-1] + res; i--; j--;

    }

    else if (dp[i - 1][j] > dp[i][j - 1]) i--;

    else j--;

  }

return res;

}

# LCA

void preprocess(int n, vector<int> t, vector<vi>& p) {

    int i, j;

    for (i=0;i<n;i++){

        for (j=0;(1<<j)<n;j++) {

            p[i][j] = -1;

        }

    }

    for (i=0;i<n;i++){ p[i][0]=t[i]; }

    for (j=1;(1<<j)<n;j++){

        for(i=0;i<n;i++){

            if (p[i][j-1]!=-1) {

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

            }

        }

    }

}

void lca(int u, int v) {

    if (h[u]<h[v]) swap(u,v);

    int logHeight = log2(h[u]);

    for (int i=logHeight;i>=0;i--){

        if (h[u]-(1<<i)>=h[v]) u = p[u][i];

    }

    if (u==v) return u;

    for (int i=logHeight;i>=0;i--){

        if (p[u][i]!=-1 && p[u][i]!=p[v][i]) {

            u = p[u][i]; v = p[v][i];

        }

    }

    return t[u];

}

# Z algorithm

vector<int> computeZ(string s) {

    int n = s.length();

    vector<int> z(n, 0);

    z[0] = n;

    int l=0, r=0;

    for (int i=1; i<n; i++){

        if (r>=i) {

            z[i] = min(z[i-l], r-i+1);

        }

        while (i+z[i]<n && s[i+z[i]]==s[z[i]]) {

            z[i]++;

        }

        if (i+z[i]-1 > r) {

            r = i+z[i]-l;

            l = i;

        }

    }

    return z;

}

# String function

string s0 ("Initial string");

string s1;

string s2(s0);

string s3(s0, 8, 3);

string s4(10, 'x'); //10 times x

string s5(10, 42);  //10 times \*

string s6(s0.begin(), s0.begin()+7);

string s7 = s1.substr(pos, len);    // O(n)

string s8 = s1.substr(pos);         // From pos to end

s0.erase(size, len);    // O(n)

s0.insert(pos, s1);     // O(n)

s0.length();

s0.replace(pos, len, s1);

int num = stoi(str, ptr, base);

string s = "This is: " + to\_string(num);

# Set function

set<int> s;

for (int i=1;i<10;i++) s.insert(10\*i);

s.erase(40);         //O(logN)

s.erase(s.begin()); //O(1)

s.clear()           //O(N)

# Vector function

vector<int> v1, v2, v3;

auto it = set\_union(v1, v1.end(), v2, v2.end(), v3);

v3.resize(it-v3);