

Team notebook

Omar Faruque, Abdullah Al Shaad, Gazi Mohaimin Iqbal

August 29, 2020

Contents		
1 mohaimin	1	
1.1 DP optimizations	1	
1.1.1 Divide and conquer optimization	1	
1.1.2 Dynamic Convex Hull trick	1	
1.1.3 SOS dp	2	
1.1.4 Semi-offline convex hull trick	2	
1.1.5 knuth optimization	2	
1.2 Data Structures	2	
1.2.1 DSU (path compression + union by size)	2	
1.2.2 mos	3	
1.2.3 sparse table (0-based indexing)	3	
1.3 FFT	3	
1.3.1 2D FFT	3	
1.3.2 FWHT	4	
1.3.3 fft by ashiqu	5	
1.4 Math	6	
1.4.1 Gaussian elimination modulo 2 (32 times faster)	6	
1.4.2 Gaussian elimination	6	
1.5 Policy based Data structures	6	
1.6 bigint	7	
1.7 bit math	8	
1.7.1 bit formula	8	
1.8 bitset	8	
1.9 combinatorics	8	
1.9.1 combinatorics	8	
1.10 flow	9	
1.10.1 Dinic with and without scaling	9	
1.10.2 Hopcroft karp	10	
1.10.3 minCostflow by dacin21(1)	10	
1.11 geometry	12	
1.11.1 Hill Climbing algorithm	12	
1.11.2 closest pair of points	12	
1.11.3 comparison of doubles	13	
1.11.4 convex hull 2D	13	
1.11.5 geo routine	13	
1.11.6 half plane intersection Radewoosh style (OFFLINE)	15	
1.12 graph	16	
1.12.1 FloydWarshall	16	
1.12.2 articulation bridges	16	
1.12.3 centroid decomposition	16	
1.12.4 djikstra	16	
1.12.5 dsu on trees (contest version)	16	
1.12.6 euler circuit and path(2 in one)	17	
1.12.7 hld	17	
1.12.8 lca	18	
1.12.9 topological sort (tarjan)	18	
1.12.10 virtual tree	19	
1.13 graph or integer sequence hashing	19	
1.13.1 graph hashing using modulo and prime(Xellos approved)	19	
1.13.2 graph hashing using xor(Xellos approved)	19	
1.14 important C++ features	20	
1.15 josephus problem	20	
1.16 matrix exponentiation	20	
1.17 number theory	21	
1.17.1 Mobius inversion	21	
1.17.2 Tonelli-Shanks(for solving quadratic congruence equation)	21	
1.17.3 bigmod	21	
1.17.4 factorial factorize	21	
1.17.5 mod inverse from 1 to N	21	
1.17.6 pollard rho	22	
1.17.7 sieve	22	
1.18 strings	22	
1.18.1 aho corasick	22	
1.18.2 manachers	23	
1.18.3 suffix array(better version by Ashiquul)	23	
1.18.4 z-algorithm	24	
1.19 ternary search	24	
2 omar bhai	25	
2.1 Data Structure	25	
2.1.1 2D Fenwick tree	25	
2.2 Hashing	25	
2.3 Number theory	25	
2.3.1 0Number theory rules	25	
2.3.2 Chinese remainder theorem	25	
2.3.3 Linear Diophantine equation	26	
2.3.4 Sum of NOD of numbers from 1 to N	26	
2.4 Rotating Callipers	26	
3 shaad bhai	27	
3.1 Intersection of Two Path	27	
1 mohaimin		
1.1 DP optimizations		
1.1.1 Divide and conquer optimization		
<hr/>		
<pre>/// Original Recurrence: dp[i][j] = min(dp[i-1][k] + C[k][j]) for k < j (** C[k+1][j] instead of C[k][j] is also possible; max instead of min is also possible)</pre>		
<pre>/// Sufficient condition: opt[i][j] <= opt[i][j+1] where opt[i][j] = smallest k that gives optimal answer for dp[i][j]</pre>		

```

/// 1 <= i <= n, 1 <= j <= m
/// Again, divide and conquer optimization can be
/// applied if it satisfies the following condition:
/// (approved by cgy4ever)
1. Quadrangle inequality:  $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$ ,  $a \leq b \leq c \leq d$ 
/// Original complexity:  $O(n*m^2)$ , Optimized
/// complexity:  $O(n*m*\lg(m))$ 
/// ** 1-based indexing.
/// the implementation is for min. for max modify the
/// code slightly.
/// to eliminate possibility of overflow reduce the
/// value of infLL

long long C(int i, int j); /// precompute C function
beforehand;
ll dp[MAXN][MAXM]; /// See if you can optimize memory
by declaring dp[2][MAXM];

void compute(int idx, int l, int r, int optl, int optr)
{
    if (l > r) return;
    int mid = (l + r) >> 1;
    pair<long long, int> best = {infLL, -1}; /// take
    -infLL for max answer.
    for (int k = optl; k <= min(mid-1, optr); k++) {
        /// take < for max answer; take C(k+1, mid) if
        required;
        if (best.F > dp[idx-1][k] + C(k, mid)) best =
        MP(dp[idx-1][k] + C(k, mid), k); /// watch
        out for possible dp[2][MAXM] memory
        optimization!
    }
    dp[idx][mid] = best.first;
    int opt = best.second;
    compute(idx, l, mid-1, optl, opt);
    compute(idx, mid+1, r, opt, optr);
}

int main() {
    int n, m;
    dp[0][0] = 0;
    for (int i = 1; i <= m; ++i) dp[0][i] = infLL;
    /// take -infLL for max answer;
    for (int i = 1; i <= n; ++i) compute(i, 1, m, i-1,
    m); /// apparently compute(i, 1, m, 0, m)
    also works, i don't know why.
    cout << dp[n][m] << endl;
}

```

1.1.2 Dynamic Convex Hull trick

```

//Insertion -  $O(\lg N)$ , query -  $O(\lg N)$ 
//Querying in empty cht container gives runtime error!
//y = mx+b;
const ll IS_QUERY = -(1LL<<62); //-INF
struct Line {
    ll m, b;
    mutable function<const Line*> succ;
    bool operator<(const Line& rhs) const {
        if (rhs.b != IS_QUERY) return m < rhs.m;
        const Line* s = succ();
        if (!s) return 0;
        ll x = rhs.m;
        return (long double)b - s->b < (long double)(s->m -
        m) * x; //add precision only if wa;
    }
};
// Keeps upper hull for maximums.
// add lines with -m and -b and return -ans to make
this code working for minimums.
struct HullDynamic : public multiset<Line> {
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;
        return (long double)(x->b - y->b)*(z->m - y->m) >=
        (long double)(y->b - z->b)*(y->m - x->m);
        //add precision only if wa;
    }
    void insert_line(ll m, ll b) {
        auto y = insert({m, b}); // insert({-m, -b})
        instead to work for minimums.
        y->succ = [=] { return next(y) == end() ? 0 :
        &*next(y); };
        if (bad(y)) { erase(y); return; }
        while (next(y) != end() && bad(next(y)))
            erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll query(ll x) {
        auto l = *lower_bound((Line) { x, IS_QUERY });
        return l.m * x + l.b; // return (-1LL)*(l.m * x +
        l.b) instead for getting minimums.
    }
};
int main() {
    HullDynamic cht;
    int T;
    cin >> T;
    while(T--) {

```

```

        cht.clear();
    }
}

1.1.3 SOS dp

// $O(N*2^N)$ 
int dp[1<<N];
for(int i = 0; i < (1<<N); ++i) dp[i] = A[i];
for(int i = 0; i < N; ++i) {
    for(int mask = 0; mask < (1<<N); ++mask) {
        if(mask & (1<<i)) dp[mask] += dp[mask^(1<<i)];
        ///checkBit, ResetBit;
    }
}

// $O(3^N)$  (Very much Easier to modify)
int dp[1<<N];
// iterate over all the masks
for (int mask = 0; mask < (1<<N); mask++) {
    dp[mask] = A[0];
    // iterate over all the subsets of the mask
    for (int i = mask; i > 0; i = (i-1) & mask) {
        dp[mask] += A[i];
    }
}

```

1.1.4 Semi-offline convex hull trick

```

//Covers all types of conditions :)
struct ConvexHullTrick {
    vector<ll> m, b;
    int ptr = 0;
    bool bad(int l1, int l2, int l3) { //add precision
    if only WA.
        return (long double)(b[l3] - b[l1])*(m[l1] - m[l2])
        <= (long double)(b[l2] - b[l1])*(m[l1] -
        m[l3]); /// (slope dec+query min), (slope
        inc+query max)
        return (long double)(b[l3] - b[l1])*(m[l1] - m[l2])
        > (long double)(b[l2] - b[l1])*(m[l1] -
        m[l3]); /// (slope dec+query max), (slope
        inc+query min)
    }
    void insert_line(ll _m, ll _b) {
        m.push_back(_m);
        b.push_back(_b);
        int s = m.size();

```

```

while(s >= 3 && bad(s-3, s-2, s-1)) {
    s--;
    m.erase(m.end()-2);
    b.erase(b.end()-2);
}
}
ll f(int i, ll x) { return m[i]*x + b[i]; }

//(slope dec+query min), (slope inc+query max) -> x
//increasing
//(slope dec+query max), (slope inc+query min) -> x
//decreasing
ll query(ll x) {
    if(ptr >= m.size()) ptr = m.size()-1;
    while(ptr < m.size()-1 && f(ptr+1, x) < f(ptr, x))
        ptr++; // f(ptr+1, x) > f(ptr, x) for max
    query
    return f(ptr, x);
}

//if there is no condition on x then simply ternary
//search on the function "f"
};

```

1.1.5 knuth optimization

```

// *** Very easy. You should be able to implement it
// on your own without looking at the implementation
// below even for a single moment;

// Original recurrence:
// dp[i][j] = min(dp[i][k] + dp[k][j]) + C[i][j]
// for i < k < j ( *** dp[i][k-1]+dp[k+1][j] is also
// possible; dp[i][k]+dp[k+1][j] is also possible;
// max instead of min is also possible )

// Sufficient condition: opt[i][j-1]
// <= opt[i][j] <= opt[i+1][j] where opt[i][j] =
// smallest k that gives optimal answer

// 1 <= i < j <= n;
// *** 1-based indexing
// Original Complexity: O(n^3), Optimized Complexity:
// O(n^2)

// Again, knuth optimization can be applied if it
// satisfies the following two conditions:
// 1. Quadrangle inequality: C[a][c]+C[b][d] <=
// C[a][d]+C[b][c], a <= b <= c <= d
// 2. Monotonicity: C[b][c] <= C[a][d], a <= b <= c <=
// d

// To eliminate the possibility of overflow reduce the
// value of infLL;

```

```

// The implementation below is for min. for max modify
// the code slightly.

int n;
int dp[500][500];
int C[500][500]; // precompute C function.
int opt[500][500];

int main() {
    for(int i = 1; i <= n; ++i) {
        dp[i][i] = C[i][i];
        opt[i][i] = i;
    }
    for(int len = 2; len <= n; ++len) {
        for(int i = 1, j = len; j <= n; ++i, ++j) {
            pair<ll, int> best = MP(infLL, -1); //take
            -infLL for max;
            for(int k = opt[i][j-1]; k <= opt[i+1][j]; ++k) {
                //take < for max; take dp[i][k]+dp[k][j] if
                //required;
                if( best.F > dp[i][k-1]+dp[k+1][j]+C[i][j] )
                    best = MP( dp[i][k-1]+dp[k+1][j]+C[i][j], k
                    );
            }
            dp[i][j] = best.F;
            opt[i][j] = best.S;
        }
    }
    cout << dp[1][n] << endl;
}

```

1.2 Data Structures

1.2.1 DSU (path compression + union by size)

```

int par[10005];
int sz[10005];
int Find(int r) { //Path compression
    if( par[r] == r ) return r;
    par[r] = Find(par[r]);
    return par[r];
}

void Union(int a, int b) { //Union by size of subtrees.
    a = Find(a);
    b = Find(b);
    if (a != b) {
        if (sz[a] < sz[b]) swap(a, b);
        par[b] = a;
        sz[a] += sz[b];
    }
}

```

```

}
int main() {
    for(int i = 0; i < 10005; ++i) {
        par[i] = i;
        sz[i] = 1;
    }
}

```

1.2.2 mos

```

//Complexity: O(sqrt(N) * (N+M))
const int mx = 3e5+5;
const int block_sz = 550; // N ~ 3e5
int freq[mx], mo_cnt = 0;
int ret[mx];
inline void add(int idx) {
    ++freq[a[idx]];
    if(freq[a[idx]] == 1) ++mo_cnt;
}
inline void del(int idx) {
    --freq[a[idx]];
    if(freq[a[idx]] == 0) --mo_cnt;
}
inline int get_ans() {
    return mo_cnt;
}

struct queries {
    int l, r, idx;
    queries() {}
    queries(int _l, int _r, int _i) : l(_l), r(_r),
    idx(_i) {}
    bool operator < (const queries &p) const {
        if(l/block_sz != p.l/block_sz) return l < p.l;
        return ((l/block_sz & 1) ? r > p.r : r < p.r);
    }
};

void mo(vector<queries> &q) {
    sort(q.begin(), q.end());
    memset(ret, -1, sizeof ret);
    // l = 1, r = 0 if 1-indexed array
    int l = 0, r = -1;
    for(auto &qq : q) {
        while(qq.l < l) add(--l);
        while(qq.r > r) add(++r);
        while(qq.l > l) del(l++);
        while(qq.r < r) del(r--);
        ret[qq.idx] = get_ans();
    }
}

```

1.2.3 sparse table (0-based indexing)

```
//1. range sum query (copied from cp-algorithms)
const int MAXN = 1e5+5;
long long arr[MAXN];
const int LOG = log2(MAXN) + 1;
long long sp[MAXN][LOG+1];
int N; //size of arr
void init() { //O(NlogN)
    for( int i = 0; i < N; ++i ) sp[i][0] = arr[i];
    for( int j = 1; j <= LOG; ++j )
        for( int i = 0; i + (1 << j) <= N; ++i )
            sp[i][j] = sp[i][j-1] + sp[i + (1 << (j - 1))][j - 1];
}
long long query( int L, int R ) { //O(logN)
    long long sum = 0;
    for( int j = LOG; j >= 0; --j ) {
        if((1 << j) <= R - L + 1) {
            sum += sp[L][j];
            L += 1 << j;
        }
    }
    return sum;
}
//2. rmq (copied from cp-algorithms)
const int MAXN = 1e5+5;
long long arr[MAXN];
const int LOG = log2(MAXN) + 1;
long long sp[MAXN][LOG+1];
int prelog[MAXN];
int N; //size of arr
void init_log() { //O(MAXN)
    prelog[1] = 0;
    for( int i = 2; i < MAXN; ++i ) prelog[i] =
        prelog[i/2] + 1;
}
void init() { //O(NlogN)
    for( int i = 0; i < N; ++i ) sp[i][0] = arr[i];
    for( int j = 1; j <= LOG; ++j )
        for( int i = 0; i + (1 << j) <= N; ++i )
            sp[i][j] = min(sp[i][j-1], sp[i + (1 << (j - 1))][j - 1]);
}
long long query(int L, int R) { //O(1)
    int j = prelog[R - L + 1];
    long long minimum = min(sp[L][j], sp[R - (1 << j) + 1][j]);
    return minimum;
}
//3. 2D rmq
const int MAXN = 1005;
```

```
const int MAXM = 1005;
long long arr[MAXN][MAXM];
const int LOGN = log2(MAXN) + 1;
const int LOGM = log2(MAXM) + 1;
long long sp[MAXN][MAXM][LOGN + 1][LOGM + 1];
int prelog[max(MAXN, MAXM)];
int N, M; //size of arr is N*M
void init_log() { //O(MAXN)
    prelog[1] = 0;
    for( int i = 2; i < max(MAXN, MAXM); ++i ) prelog[i] =
        prelog[i/2] + 1;
}
void init() { //O(N*M*logN*logM)
    for( int i = 0; i < N; ++i )
        for( int j = 0; j < M; ++j ) sp[i][j][0][0] =
            arr[i][j];
    for( int k = 1; k <= LOGN; ++k ) {
        for( int i = 0; i + (1 << k) <= N; ++i ) {
            for( int j = 0; j < M; ++j ) {
                sp[i][j][k][0] = min(sp[i][j][k - 1][0], sp[i + (1 << (k - 1))][j][k - 1][0]);
            }
        }
    }
    for( int l = 1; l <= LOGM; ++l ) {
        for( int k = 0; k <= LOGN; ++k ) {
            for( int i = 0; i + (1 << k) <= N; ++i ) {
                for( int j = 0; j + (1 << l) <= M; ++j ) {
                    sp[i][j][k][l] = min(sp[i][j][k][l - 1], sp[i][j + (1 << (l - 1))][k][l - 1]);
                }
            }
        }
    }
}
long long query( int r1, int c1, int r2, int c2 ) {
    //O(1)
    int a = prelog[(r2 - r1) + 1];
    int b = prelog[(c2 - c1) + 1];
    return min(min(sp[r1][c1][a][b], sp[r2 - (1 << a) + 1][c1][a][b]), min(sp[r1][c2 - (1 << b) + 1][a][b], sp[r2 - (1 << a) + 1][c2 - (1 << b) + 1][a][b]));
}
```

1.3 FFT

1.3.1 2D FFT

```
/**
Rule of thumb for precision issues- (by Errichto)
```

Well written FFT has an error that depends mainly on the magnitude of numbers in the answer, so it works if the resulting polynomial doesn't have numbers bigger than 10^{15} for doubles and 10^{18} for long doubles. You can measure the error by finding max of $\text{abs}(\text{xllround}(x))$ over all x in the answer. If it's smaller than 0.2, you're good and rounding will be fine.

```
/**
//No need to clear anything at the beginning of a test case
// 2D FFT of a 2D polynomial ~ O(N*M*(lgN + lgM))
// product of two 2D polynomials ~ O(N*M*(lgN + lgM))
struct FFT_2D {
    struct node {
        double x,y;
        node() {}
        node(double a, double b): x(a), y(b) {}
        node operator + (const node &a) const {return
            node(this->x+a.x,this->y+a.y);}
        node operator - (const node a) const {return
            node(this->x-a.x,this->y-a.y);}
        node operator * (const node a) const {return
            node(this->x*a.x-this->y*a.y,this->x*a.y+a.x*this->y);}
    };

    int M[2];
    vector<vector<node>> A, B;
    vector<node> w[2][2];
    vector<int> rev[2];
    long double pi;
    FFT_2D() {
        pi = 3.1415926535897932384;
    }
    void init(int n, int m) {
        M[0] = 1, M[1] = 1;
        while(M[0] < n) M[0] <= 1;
        M[0] <= 1;
        while(M[1] < m) M[1] <= 1;
        M[1] <= 1;
        A.resize(M[0], vector<node>(M[1]));
        B.resize(M[0], vector<node>(M[1]));
        for( int z = 0; z < 2; ++z ) {
            w[z][0].resize(M[z]);
            w[z][1].resize(M[z]);
            rev[z].resize(M[z]);
            for( int i=0; i<M[z]; i++) {
                int j=i,y=0;
                for( int x=1; x<M[z]; x<=1,j>=1) (y<=1)?j+=1:j-=1;
                rev[z][i]=y;
            }
            for( int i=0; i<M[z]; i++) {
```

```

w[z][0][i] = node(
    cos(2*pi*i/M[z]),sin(2*pi*i/M[z]) );
w[z][1][i] = node(
    cos(2*pi*i/M[z]),-sin(2*pi*i/M[z]) );
}
}
void ftransform_2D( vector<vector<node>> &A, int p ) {
    for( int z = 0; z < M[0]; z++ ) {
        for( int i=0; i<M[1]; i++ )
            if (i<rev[1][i])
                swap(A[z][i],A[z][rev[1][i]]);
        for( int i=1; i<M[1]; i<=1 )
            for( int j=0,t=M[1]/(i<<1); j<M[1]; j+=i<<1 )
                for( int k=0,l=0; k<i; k++,l+=t ) {
                    node x=A[z][i+j+k]*w[1][p][l];
                    node y=A[z][j+k];
                    A[z][j+k]=y+x;
                    A[z][j+k+i]=y-x;
                }
        if (p)
            for( int i=0; i<M[1]; i++ )
                A[z][i].x/=M[1], A[z][i].y/=M[1];
    }
    for( int z = 0; z < M[1]; z++ ) {
        for( int i=0; i<M[0]; i++ )
            if (i<rev[0][i])
                swap(A[i][z],A[rev[0][i]][z]);
        for( int i=1; i<M[0]; i<=1 )
            for( int j=0,t=M[0]/(i<<1); j<M[0]; j+=i<<1 )
                for( int k=0,l=0; k<i; k++,l+=t ) {
                    node x=w[0][p][l]*A[i+j+k][z];
                    node y=A[j+k][z];
                    A[j+k][z]=y+x;
                    A[j+k+i][z]=y-x;
                }
        if (p)
            for( int i=0; i<M[0]; i++ )
                A[i][z].x/=M[0];
    }
}
}
//2degree polynomial P * 2degree polynomial Q =
//2degree polynomial res
//Degree of P is x1,x2; Degree of Q is y1, y2;
//Degree of res is x1+y1, x2+y2;
void multiply_2D( vector<vector<long long>> &P,
    vector<vector<long long>> &Q, vector<vector<long
    long>> &res ) {
    init( max(P.size(), Q.size()), max(P[0].size(),
        Q[0].size()) );
    for( int i = 0; i < M[0]; i++ )
        for( int j = 0; j < M[1]; j++ )

```

```

        A[i][j].x = A[i][j].y = B[i][j].x = B[i][j].y =
            0;
        for( int i = 0; i < P.size(); i++ )
            for( int j = 0; j < P[i].size(); j++ )
                A[i][j].x = P[i][j];
        for( int i = 0; i < Q.size(); i++ )
            for( int j = 0; j < Q[i].size(); j++ )
                B[i][j].x = Q[i][j];
        ftransform_2D(A, 0);
        ftransform_2D(B, 0);
        for( int i = 0; i < M[0]; i++ )
            for( int j = 0; j < M[1]; j++ )
                A[i][j] = A[i][j]*B[i][j];
        ftransform_2D(A, 1);
        res.resize(M[0], vector<ll>(M[1]));
        for( int i = 0; i < M[0]; i++ )
            for( int j = 0; j < M[1]; j++ )
                res[i][j] = round(A[i][j].x);
    }
}fft_2D;

int main() {
    // a*b = c
    // Degree of a = (n1-1, n2-1), Degree of b = (m1-1,
        m2-1), Degree of c = (n1+m1-2, n2+m2-2);
    vector<vector<ll>> a, b, c;
    int n1, n2, m1, m2;
    cin >> n1 >> n2 >> m1 >> m2;
    a.resize(n1, vector<ll>(n2)), b.resize(m1,
        vector<ll>(m2)), c.resize(n1+m1-1,
        vector<ll>(n2+m2-1));
    for( int i = 0; i < n1; ++i )
        for( int j = 0; j < n2; ++j ) cin >> a[i][j];
    for( int i = 0; i < m1; ++i )
        for( int j = 0; j < m2; ++j ) cin >> b[i][j];
    fft_2D.multiply(a, b, c); // product of a and b is
        kept in c;
}

```

1.3.2 FWHT

```

//O(NlogN)
//Doubles are never used as in FFT. Always long long
    is used(modulo or not).

#define MOD 1000000007

#define OR 0
#define AND 1
#define XOR 2
struct FWHT {

```

```

void walsh_transform(vector<ll> &p, bool inv, int
    flag) {
    int n = p.size(); assert((n&(n-1))==0);
    for (int len=1; 2*len<=n; len <= 1) {
        for (int i = 0; i < n; i += len+len){
            for (int j = 0; j < len; j++) { //Don't forget
                to include p[i+j] = (p[i+j]+MOD)%MOD,
                p[i+len+j] = (p[i+len+j]+MOD)%MOD at the
                end of this for loop(inside of course) when
                required.
                ll u = p[i+j], v = p[i+len+j];
                if( (flag == XOR) ) p[i+j]=u+v, p[i+len+j]=u-v;
                if( (flag == AND) && !inv ) p[i+j]=v,
                p[i+len+j]=u+v;
                if( (flag == OR) && !inv ) p[i+j]=u+v,
                p[i+len+j]=u;
                if( (flag == AND) && inv ) p[i+j]= -u+v,
                p[i+len+j]=u;
                if( (flag == OR) && inv ) p[i+j]=v,
                p[i+len+j]=u-v;
            }
        }
    }
    if(inv && (flag == XOR)) for(int i=0;i<n;i++)
        p[i]/=n; //Don't forget modular inverse n
        when required
}
// For i = 0 to n - 1, j = 0 to n - 1
// v[i flag j] += a[i] * b[j]
vector<ll> convo(vector<ll> &a, vector<ll> &b, int
    flag = XOR) {
    int n = 1, sz = max(a.size(), b.size());
    while(n<sz) n*=2;
    a.resize(n); b.resize(n); vector<ll>res(n, 0);
    walsh_transform(a, 0, flag); walsh_transform(b, 0,
        flag);
    for(int i=0;i<n;i++) res[i] = a[i] * b[i]; //
        Don't forget modular multiplication when
        required.
    walsh_transform(res, 1, flag);
    return res;
}
//compute A^k where A?A == A convolution A
vector<ll> pow(vector<ll> &A, ll k, int flag = XOR) {
    int n = 1, sz = A.size();
    while(n<sz) n *= 2;
    A.resize(n);
    walsh_transform(A, 0, flag);
    for(int i=0; i<n; i++) A[i]=expo(A[i], k, MOD);
    //MOD = infLL if there is no modulo
    required.(trix, modern problems require modern
    solutions.)
    walsh_transform(A, 1, flag);
}

```

```

    return A;
}
}fwht;

int main() {
    vector<ll> vec(6);
    vector<ll> temp = vec;
    vector<ll> store1 = fwht.convo(temp, temp, XOR);
    //size of store1 will always be a power of 2.
    temp = vec;
    vector<ll> store2 = fwht.pow(temp, 4, XOR); //size
    of store2 will always be a power of 2.
}

```

1.3.3 fft by ashiquel

```

/**
Rule of thumb for precision issues- (by Errichto)
Well written FFT has an error that depends mainly on
the magnitude of numbers in the answer, so it
works if the resulting polynomial doesn't have
numbers bigger than 1015 for doubles and 1018
for long doubles. You can measure the error by
finding max of abs(xllround(x)) over all x in the
answer. If it's smaller than 0.2, you're good and
rounding will be fine.
**/
//No need to clear anything at the beginning of a test
case

/// FFT
struct FFT {
    struct node {
        double x,y;
        node() {}
        node(double a, double b): x(a), y(b) {}
        node operator + (const node &a) const {return
            node(this->x+a.x,this->y+a.y);}
        node operator - (const node a) const {return
            node(this->x-a.x,this->y-a.y);}
        node operator * (const node a) const {return
            node(this->x*a.x-this->y*a.y,this->x*a.y+a.x*this->y);}
    };

    int M;
    vector<node> A, B, w[2];
    vector<int> rev;
    long double pi;
    FFT() {
        pi = 3.1415926535897932384;
    }
}

```

```

void init(int n) {
    M = 1;
    while(M < n) M <= 1;
    M <= 1;
    A.resize(M);
    B.resize(M);
    w[0].resize(M);
    w[1].resize(M);
    rev.resize(M);
    for (int i=0; i<M; i++) {
        int j=i,y=0;
        for (int x=1; x<M; x<=(1,j)>=1) (y<=(1)+=j&1;
        rev[i]=y;
    }
    for (int i=0; i<M; i++) {
        w[0][i] = node( cos(2*pi*i/M),sin(2*pi*i/M) );
        w[1][i] = node( cos(2*pi*i/M),-sin(2*pi*i/M) );
    }
}

void ftransform( vector<node> &A, int p ) {
    for (int i=0; i<M; i++)
        if (i<rev[i])
            swap(A[i],A[rev[i]]);
    for (int i=1; i<M; i<=(1)
        for (int j=0,t=M/(i<=1); j<M; j+=i<=1)
            for (int k=0,l=0; k<i; k++,l+=t) {
                node x=w[p][l]*A[i+j+k];
                node y=A[j+k];
                A[j+k]=y+x;
                A[j+k+i]=y-x;
            }
    if (p)
        for (int i=0; i<M; i++)
            A[i].x/=M;
}

// multiply P*Q and keeps the result in res
//degree of P is n and degree of Q is m
//P, Q is given in standard power form, in increasing
void multiply( vector<long long> &P, vector<long
long> &Q, vector<long long> &res) {
    init( max(P.size(),Q.size()) );
    for( int i = 0; i < M; i++ )
        A[i].x = A[i].y = B[i].x = B[i].y = 0;
    for( int i = 0; i < P.size(); i++ )
        A[i].x = P[i];
    for( int i = 0; i < Q.size(); i++ )
        B[i].x = Q[i];
    ftransform(A,0);
    ftransform(B,0);
    for (int k=0; k<M; k++)
        A[k] = A[k]*B[k];
    ftransform(A,1);
    res.resize(M);
}

```

```

    for( int i = 0; i < M; i++ )
        res[i] = round(A[i].x);
    }
}fft;

int main() {
    // a*b = c
    // Degree of a = n-1, Degree of b = m-1, Degree of c
    = n+m-2;
    vector<ll> a, b, c;
    int n, m;
    cin >> n >> m;
    a.resize(n), b.resize(m), c.resize(n+m-1);
    for( int i = 0; i < n; ++i ) cin >> a[i];
    for( int i = 0; i < m; ++i ) cin >> b[i];
    fft.multiply(a, b, c); // product of a and b is kept
    in c;
}

```

1.4 Math

1.4.1 Gaussian elimination modulo 2 (32 times faster)

```

// calculates 32 times faster for modulo 2
/*
n rows/equations, m+1 columns, m variables
calculates determinant, rank and ans[] -> value for
variables
returns {0, 1, INF} -> number of solutions
gives truly row echelon form.(sort of Identity
matrixish, but the independent column/variables
may have non-zero values at the upper rows.)
*/

const double EPS = 1e-9;
#define INF 1000000000
#define MAX 105
int Rank; //Rank means number of non-zero rows in row
echelon form

int Det;
int gauss(vector<bitset<MAX>> a, bitset<MAX> &ans,
    int n, int m) { // number of variables must be
    defined!
    ans.reset(); //reset all bits;
    vector<int> where(m, -1);
    Det = 1, Rank = 0;
    for(int col = 0, row = 0; col < m && row < n; ++col) {
        int sel = row;

```



```

for(int i = row; i < n; ++i) if(a[i][col]) { sel =
    i; break; }
if(!a[sel][col]) { Det = 0; continue; }
swap(a[sel], a[row]);
if(row != sel) Det = -Det; ///add mod? (idk, i made
    this up. munim vai kept this line as it is. So
    change only if WA)
Det &= a[row][col];
where[col] = row;

for(int i = 0; i < n; ++i) if (i != row &&
    a[i][col] > 0) a[i] ^= a[row];
++row, ++Rank;
}

for(int i = 0; i < m; ++i) ans[i] = (where[i] == -1)
    ? 0 : a[where[i]][m];
for(int i = Rank; i < n; ++i) if(a[i][m]) return 0;
for(int i = 0; i < m; ++i) if(where[i] == -1) return
    INF;
return 1;
}

```

1.4.2 Gaussian elimination

```

/*
    n rows/equations, m+1 columns, m variables
    calculates determinant, rank and ans[] -> value for
    variables
    returns {0, 1, INF} -> number of solutions
    gives truly row echelon form. (sort of Identity
    matrixish, but the independent column/variables
    may have non-zero values at the upper rows.)
*/
///For "gaussian elimination modulo version"-
1. add big mod function to the top.
2. just do all the calculations in mod(determinant
    too. but not rank).
3. change fabs() to abs() maybe??
4. I have add comments to the code to help with
    "steps- 2 and 3".
5. change all doubles to long long as all the
    calculations are done in long long. omit EPS.
*/
const double EPS = 1e-9;
#define INF 100000000
int Rank; ///Rank means number of non-zero rows in
    row-echelon form
double Det;

```

```

int gauss( vector<vector<double>> a, vector<double>
    &ans ) {
    int n = (int)a.size(), m = (int)a[0].size()-1;
    ans.clear(); ans.resize(m);
    vector<int> where(m, -1);
    Det = 1.0, Rank = 0;
    for(int col = 0, row = 0; col < m && row < n; ++col) {
        int sel = row;
        for(int i = row+1; i < n; ++i)
            if(fabs(a[i][col]) > fabs(a[sel][col])) sel = i;
        ///For modulo version: if(abs(a[i][col]) >
            abs(a[sel][col]))
        if(fabs(a[sel][col]) < EPS) { Det = 0.0; continue; }
        ///For modulo version: if(!a[sel][col])
        for(int j = 0; j <= m; ++j) swap(a[sel][j],
            a[row][j]);
        if(row != sel) Det = -Det; ///For modulo version:
            add mod? (idk, i made this up. munim vai kept
            this line as it is. So change only if WA)
        Det *= a[row][col]; ///For modulo version: mod mul
        where[col] = row;

        double s = (1.0 / a[row][col]); ///For modulo
            version: mod inverse
        for(int j = 0; j <= m; ++j) a[row][j] *= s; ///For
            modulo version: mod mul
        for(int i = 0; i < n; ++i) if (i != row &&
            fabs(a[i][col]) > EPS) { ///For modulo
                version: if (i != row && a[i][col] > 0)
            double t = a[i][col];
            for(int j = 0; j <= m; ++j) a[i][j] -= a[row][j]
                * t; ///For modulo version: mod mul and mod
                    sub
            }
            ++row, ++Rank;
        }

        for(int i = 0; i < m; ++i) ans[i] = (where[i] == -1)
            ? 0.0 : a[where[i]][m];
        for(int i = Rank; i < n; ++i) if(fabs(a[i][m]) > EPS)
            return 0; ///For modulo version: if(a[i][m])
        for(int i = 0; i < m; ++i) if(where[i] == -1) return
            INF;
        return 1;
    }
}

```

1.5 Policy based Data structures

```

#include<bits/stdc++.h>
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>

```

```

using namespace __gnu_pbds;
using namespace std;

template <typename T>
using ordered_set = tree<T, null_type, less<T>,
    rb_tree_tag, tree_order_statistics_node_update>;
///set in ascending order (the typical one)
template <typename T>
using ordered_set_of_pairs = tree<pair<T, size_t>,
    null_type, less<pair<T, size_t>>, rb_tree_tag,
    tree_order_statistics_node_update>; ///set of
    pairs in ascending order (the typical one)
template <typename T>
using ordered_set_desc = tree<T, null_type, greater<T>,
    rb_tree_tag, tree_order_statistics_node_update>;
///set in descending order
template <typename T>
using ordered_set_of_pairs_desc = tree<pair<T, size_t>,
    null_type, greater<pair<T, size_t>>, rb_tree_tag,
    tree_order_statistics_node_update>; ///set of
    pairs in descending order

#define optimize()
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
#define endl '\n'

#define MP make_pair
#define F first
#define S second
#define PB push_back

ordered_set<int> st1;
ordered_set_of_pairs<int> st2;
ordered_set_desc<int> st3;
ordered_set_of_pairs_desc<int> st4;

int main() {
    optimize();
    for( int i = 0; i < 10; ++i ) st1.insert(i);
    cout << st1.order_of_key(2) << endl; ///how many
        elements in st1 less than 2? (output: 2)
    cout << *st1.find_by_order(5) << endl << endl;
    ///what is the 5th minimum element in st1? (0th
        based indexing) (output: 5)
    for( int i = 0; i < 10; ++i ) st2.insert(MP(i, i));
    cout << st2.order_of_key(MP(2, 3)) << endl;
    ///output: 3
    cout << st2.order_of_key(MP(2, 2)) << endl;
    ///output: 2
    cout << st2.order_of_key(MP(3, 2)) << endl;
    ///output: 3
}

```

```

cout << st2.order_of_key(MP(3, -1)) << endl;
//output: 4 (i know, you were expecting 3. but
//giving negative numbers as second element gives
//unexpected results.)
cout << (*st2.find_by_order(5)).F << " " <<
(*st2.find_by_order(5)).S << endl << endl;
//output: 5 5
}

```

1.6 bigint

```

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

#define optimize()
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
#define endl '\n'

struct Bigint {
    // representations and structures
    string a; // to store the digits
    int sign; // sign = -1 for negative numbers, sign = 1
    // otherwise
    // constructors
    Bigint() {} // default constructor
    Bigint( string b ) { (*this) = b; } // constructor
    // for string
    // some helpful methods
    int size() { // returns number of digits
        return a.size();
    }
    Bigint inverseSign() { // changes the sign
        sign *= -1;
        return (*this);
    }
    Bigint normalize( int newSign ) { // removes leading
        // 0, fixes sign
        for( int i = a.size() - 1; i > 0 && a[i] == '0';
            i-- )
            a.erase(a.begin() + i);
        sign = ( a.size() == 1 && a[0] == '0' ) ? 1 :
            newSign;
        return (*this);
    }
    // assignment operator
    void operator = ( string b ) { // assigns a string to
        // Bigint
        a = b[0] == '-' ? b.substr(1) : b;
        reverse( a.begin(), a.end() );
        this->normalize( b[0] == '-' ? -1 : 1 );
    }
}

```

```

// conditional operators
bool operator < ( const Bigint &b ) const { // less
    // than operator
    if( sign != b.sign ) return sign < b.sign;
    if( a.size() != b.a.size() )
        return sign == 1 ? a.size() < b.a.size() :
            a.size() > b.a.size();
    for( int i = a.size() - 1; i >= 0; i-- ) if( a[i]
        != b.a[i] )
        return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
    return false;
}
bool operator == ( const Bigint &b ) const { //
    // operator for equality
    return a == b.a && sign == b.sign;
}
// mathematical operators
Bigint operator + ( Bigint b ) { // addition operator
    // overloading
    if( sign != b.sign ) return (*this) -
        b.inverseSign();
    Bigint c;
    for(int i = 0, carry = 0; i < a.size() || i < b.size()
        || carry; i++ ) {
        carry += (i < a.size() ? a[i] - 48 : 0) + (i < b.a.size() ?
            b.a[i] - 48 : 0);
        c.a += (carry % 10 + 48);
        carry /= 10;
    }
    return c.normalize(sign);
}
Bigint operator - ( Bigint b ) { // subtraction
    // operator overloading
    if( sign != b.sign ) return (*this) +
        b.inverseSign();
    int s = sign; sign = b.sign = 1;
    if( (*this) < b ) return ((b -
        (*this)).inverseSign()).normalize(-s);
    Bigint c;
    for( int i = 0, borrow = 0; i < a.size(); i++ ) {
        borrow = a[i] - borrow - (i < b.size() ? b.a[i] :
            48);
        c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
        borrow = borrow >= 0 ? 0 : 1;
    }
    return c.normalize(s);
}
Bigint operator * ( Bigint b ) { // multiplication
    // operator overloading
    Bigint c("0");
    for( int i = 0, k = a[i] - 48; i < a.size(); i++, k
        = a[i] - 48 ) {

```

```

        while(k-- > 0) c = c + b; // ith digit is k, so, we
        // add k times
        b.a.insert(b.a.begin(), '0'); // multiplied by 10
    }
    return c.normalize(sign * b.sign);
}
Bigint operator / ( Bigint b ) { // division operator
    // overloading
    if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= (
        b.a[0] - 48 );
    Bigint c("0"), d;
    for( int j = 0; j < a.size(); j++ ) d.a += "0";
    int dSign = sign * b.sign; b.sign = 1;
    for( int i = a.size() - 1; i >= 0; i-- ) {
        c.a.insert( c.a.begin(), '0' );
        c = c + a.substr( i, 1 );
        while( !( c < b ) ) c = c - b, d.a[i]++;
    }
    return d.normalize(dSign);
}
Bigint operator % ( Bigint b ) { // modulo operator
    // overloading
    if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= (
        b.a[0] - 48 );
    Bigint c("0");
    b.sign = 1;
    for( int i = a.size() - 1; i >= 0; i-- ) {
        c.a.insert( c.a.begin(), '0' );
        c = c + a.substr( i, 1 );
        while( !( c < b ) ) c = c - b;
    }
    return c.normalize(sign);
}
// output method
void print() {
    if( sign == -1 ) cout << '-';
    for( int i = a.size() - 1; i >= 0; i-- ) cout <<
        a[i];
}
};

int main() {
    optimize();
    Bigint a, b, c; // declared some Bigint variables
    // taking Bigint input //
    string input; // string to take input
    cin >> input; // take the Big integer as string
    a = input; // assign the string to Bigint a
    cin >> input; // take the Big integer as string
    b = input; // assign the string to Bigint b
}

```



```
// Using mathematical operators //
////////////////////////////////////
c = a + b; // adding a and b
c.print(); // printing the Bigint
cout << endl; // newline
c = a - b; // subtracting b from a
c.print(); // printing the Bigint
cout << endl; // newline
c = a * b; // multiplying a and b
c.print(); // printing the Bigint
cout << endl; // newline
c = a / b; // dividing a by b
c.print(); // printing the Bigint
cout << endl; // newline
c = a % b; // a modulo b
c.print(); // printing the Bigint
cout << endl; // newline
////////////////////////////////////
// Using conditional operators //
////////////////////////////////////
if( a == b ) puts("equal"); // checking equality
else puts("not equal");
if( a < b ) puts("a is smaller than b"); // checking
less than operator
return 0;
}
```

1.7 bit math

1.7.1 bit formula

```
//& = bitwise and
//| = bitwise or
//^ = bitwise xor
1. a+b = (a^b) + 2*(a&b)
2. a+b = (a|b) + (a&b)
```

1.8 bitset

```
int main() {
    bitset<5005> bss[12] , now ;

    bss[i].set(j); // sets jth bit of i'th bitset
    now = bss[i]&bss[j]; // can do logical operation
    betwn two bitset
    int here = now.count(); // counts number of 1s in the
    bitset
}
```

```
now[i] = 1; // sets i-th bit from right ( 0 based) to
1
now.all() ; // check if all the bits are set
now.any() ; // checks if any bits are set
now.none() ; //checks if none of the bits are set

bitset<5005> bs1, bs2, bs3; //Complexity of declaring
bitsets is O(N/32)

bs3 = bs1 ^ bs2 (Complexity is O(N/32))
bs3 = bs1 & bs2 (Complexity is O(N/32))
bs3 = bs1 | bs2 (Complexity is O(N/32))
bs3 = bs2 (Complexity is O(N/32))
bs3 = ~bs3 // inverse all bits.
bs3.Find_next(idx); (Complexity is O(N/32)) //Finds
the position of set bit after idx;
bs3.Find_First(); (Complexity is O(N/32)) //Finds the
first set bit; return bs3.size() if there is no
set bit;
bs3.reset(); //rests all bits
bs3.set(); //set all bits
bs3.flip(); //flip all bits
bs3 = ~bs3; //flip all bits
cout<<bb.to_ulong()<<"\n"; // to unsigned long long
int
return 0;
}
```

1.9 combinatorics

1.9.1 combinatorics

```
1. nCr = nC(r-1) * ((n-r+1)/r)
2. nCr = (n-1)Cr + (n-1)C(r-1)
3. nC0+nC2+nC4+.... = nC1+nC3+nC5.... = 2^(n-1)
4. summation of x*nCx such that (1 <= x <= n or 0 <= x
<= n) = n*2^(n-1). (easy to prove).

1. Catalan Number(1, 1, 2, 5, 14, 42, 132, 429, 1430,
4862, 16796, 58786, 208012, 742900, 2674440,
9694845, 35357670, 129644790, 477638700,
1767263190)
1.1 Catn = (1/(n+1))*((2n)Cn)
1.2 Catn = summation of Cat(k-1)*Catk(n-k) such that
1 <= k <= n
1.3 Cat(n+1) = ((2*(2n+1))/(n+2)) * Catn
1.4 Catn = ((2*(2*(n-1)+1))/((n-1)+2)) * Cat(n-1)
```

1.10 flow

1.10.1 Dinic with and without scaling

```
/**
Dinic algorithm with scaling and not scaling.
Complexity with scaling : O(VE * lg(U)). here, U is the
maximum capacity
Complexity without scaling : O((V^2)E)
/////
Complexity in bipartite graph matching: O(E*sqrt(V)).
Complexity in unit graph: O(E*sqrt(V)). (but, Chilli
says it is wrong and actual complexity is
O(E*min(V^(2/3), E^(1/2)))) ).
A unit network is a network in which all the edges have
unit capacity, and for any vertex except s and t
either incoming or outgoing edge is unique. That's
exactly the case with the network we build to
solve the maximum matching problem with flows. So,
Dinic gives similar performance to hopcroft karp
algorithm incase of maximum bipartite matching as
it has the same time complexity.
/////
1. Works on directed graph
2. Works on undirected graph
3. Works on multi-edge(directed/undirected) graph
4. Works on self-loop(directed/undirected) graph
```

Can find the actual flow.

Can find the non-maxflow upto a certain value

```
/////
Implement it on your own. very easy. just a simple O(n)
dfs.
```

Can find minimum cut sets(A and B).

A contains source itself and the nodes that are reachable from source using non-saturated edges. B contains sink and the nodes that are not reachable from source using non-saturated edges.

Value of minimum cut capacity is summation of the per_cap of all edges u->v such that u belongs to set A, v belongs to set B.

Value of minimum cut flow is = (summation of the flow of all edges u->v such that u belongs to set A, v belongs to set B) - (summation of the flow of all edges v->u such that v belongs to set B, u belongs to set A)

Value of minimum cut capacity == maxflow.

To find the minimum cut sets(A and B) first find the maxflow. Then, we apply dfs from the source. This dfs will be such that- only

the nodes of set A will eventually be marked visited.
 Suppose we have reached in node u, that means u belongs to set A. Now,
 If (flow of edge u->v) < (per_cap of edge u->v) then v belongs to set A.
 else if (flow of edge u->v) == (per_cap of edge u->v) then v belongs to set B.

To find the minimum cut sets with minimum cardinality, first multiply (E+1) with all the edges. Then add 1 to all the edges. Then run the maxflow algorithm.

```

/////
**/
#include <bits/stdc++.h>
using namespace std;
#define optimize()
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
#define endl '\n'
template <class flow_t> struct Dinic { //int/long long;
    const static bool SCALING = true; // non-scaling =
        V^2E, Scaling=VElog(U) with higher constant
    long long lim = 1;
    const flow_t INF = numeric_limits<flow_t>::max();
    flow_t K;
    bool K2 = false;
    struct edge {
        int to, rev;
        flow_t cap, flow, per_cap;
    };
    int s, t;
    vector<int> level, ptr;
    vector< vector<edge> > adj;
    Dinic(int NN) : s(NN-2), t(NN-1), level(NN), ptr(NN),
        adj(NN) {}
    void flow_limit( flow_t val ) { //non-maxflow upto K.
        K2 = true;
        K = val;
    }
    void addEdge(int a, int b, flow_t cap, bool
        isDirected = true) {
        adj[a].push_back({b, (int)adj[b].size(), cap, 0,
            cap});
        adj[b].push_back({a, (int)adj[a].size() - 1,
            isDirected ? 0 : cap, 0, isDirected ? 0 :
            cap});
    }
    bool bfs() {
        queue<int> q({s});
        fill( level.begin(), level.end(), -1 );
        level[s] = 0;
        while (!q.empty() && level[t] == -1) {
            int v = q.front();
            q.pop();

```

```

        for (auto e : adj[v]) {
            if (level[e.to] == -1 && e.flow < e.cap &&
                (!SCALING || e.cap - e.flow >= lim)) {
                q.push(e.to);
                level[e.to] = level[v] + 1;
            }
        }
    }
    return level[t] != -1;
}
flow_t dfs(int v, flow_t flow) {
    if (v == t || !flow) return flow;
    for (; ptr[v] < adj[v].size(); ptr[v]++) {
        edge &e = adj[v][ptr[v]];
        if (level[e.to] != level[v] + 1) continue;
        if (flow_t pushed = dfs(e.to, min(flow, e.cap -
            e.flow))) {
            e.flow += pushed;
            adj[e.to][e.rev].flow -= pushed;
            return pushed;
        }
    }
    return 0;
}
flow_t max_flow(int source, int sink) {
    s = source, t = sink;
    long long flow = 0;
    for (lim = SCALING ? (1LL << 30) : 1; lim > 0; lim
        >= 1) { //Here, lim = SCALING?(U):1 ; Here U
            is an int/long long strictly greater than the
            max capacity ;
            while (bfs()) {
                fill( ptr.begin(), ptr.end(), 0 );
                while (flow_t pushed = dfs(s,
                    ((K2==true)?K:INF))) {
                    flow += pushed;
                    if(K2) {
                        K -= pushed;
                        if( K == 0 ) break;
                    }
                }
                if( K2 && (K == 0) ) break;
            }
        }
    }
    return flow;
}
vector<pair<pair<int,int>,long long>> getActualFlow()
{
    vector<pair<pair<int,int>, long long>> vec;
    for( int i = 0; i < adj.size(); ++i ) {
        for( int j = 0; j < adj[i].size(); ++j ) {
            if( adj[i][j].flow > 0 ) {

```

```

                vec.push_back( make_pair(make_pair(i,
                    adj[i][j].to), adj[i][j].flow) );
            }
        }
    }
    return vec;
}
};
int main() {
    optimize();
    int T;
    cin >> T;
    for( int test = 1; test <= T; ++test ) {
        int N, M, s, t; // no. of nodes; no. of edges;
        source; sink;
        cin >> N >> M >> s >> t;
        Dinic<int> fl(N+1); //for long long change int to
            long long; no. of nodes+1;
        for( int i = 1; i <= M; ++i ) {
            int u, v, w;
            cin >> u >> v >> w;
            fl.addEdge(u, v, w); //Directed graph;
            //fl.addEdge(u, v, w, false); //Undirected graph;
        }
        //fl.flow_limit(10); //non-maxflow upto a specific
            value;
        cout << fl.max_flow(s, t) << endl;
        vector<pair<pair<int,int>,long long>> vec =
            fl.getActualFlow();
        cout << vec.size() << endl;
        for( auto xx : vec ) {
            cout << xx.first.first << " " << xx.first.second
                << " " << xx.second << endl; //node; node;
                flow;
        }
    }
    return 0;
}

```

1.10.2 Hopcroft karp

```

//Complexity: O(sqrt(V) * E), constant may be a bit
    high.
//Works on self loops.
//if( h.R[u] != -1 ) then u is situated in left part,
    h.R[u] is situated in right part;
//if( h.L[u] != -1 ) then u is situated in right part,
    h.L[u] is situated in left part;
struct Hopcroft_karp {
    int n;
    vector< vector<int> > edge;

```

```

vector<int> dis, parent, L, R;
vector<int> Q;
Hopcroft_karp(int n_) : n(n_), edge(n_+1), dis(n_+1),
    parent(n_+1), L(n_+1), R(n_+1), Q(n_+1) {};
void add_edge( int u, int v ) {
    edge[u].push_back(v);
}
bool dfs(int i) {
    int len = edge[i].size();
    for (int j = 0; j < len; j++) {
        int x = edge[i][j];
        if (L[x] == -1 || (parent[L[x]] == i)) {
            if (L[x] == -1 || dfs(L[x])) {
                L[x] = i;
                R[i] = x;
                return (true);
            }
        }
    }
    return false;
}
bool bfs() {
    int x, f = 0, l = 0;
    fill( dis.begin(), dis.end(), -1 );
    for (int i = 1; i <= n; i++) {
        if (R[i] == -1) {
            Q[l++] = i;
            dis[i] = 0;
        }
    }
    while (f < l) {
        int i = Q[f++];
        int len = edge[i].size();
        for (int j = 0; j < len; j++) {
            x = edge[i][j];
            if (L[x] == -1) return true;
            else if (dis[L[x]] == -1) {
                parent[L[x]] = i;
                dis[L[x]] = dis[i] + 1;
                Q[l++] = L[x];
            }
        }
    }
    return false;
}
int matching() {
    int counter = 0;
    fill( L.begin(), L.end(), -1 );
    fill( R.begin(), R.end(), -1 );
    while (bfs()) {
        for (int i = 1; i <= n; i++) {
            if (R[i] == -1 && dfs(i)) counter++;
        }
    }
}

```

```

    }
    return counter;
}
};
int main() {
    int n, m; //no. of nodes; no. of edges;
    cin >> n >> m;
    Hopcroft_karp h(n);
    for( int i = 0; i < m; ++i ) {
        int u, v;
        cin >> u >> v;
        /** //Undirected:
            h.add_edge(u, v);
            h.add_edge(v, u);
        */
        /**//Directed:
            h.add_edge(u, v);
        */
        /** //Undirected:
            int ans = h.matching()/2; //we divide by 2 because
            incase of undirected graphs matching() returns
            the number of nodes involved in maximum matching;
        */
        /**//Directed:
            int ans = h.matching();
            cout << ans << endl;
            //print the actual maximum matching
            for( int i = 1; i <= n; ++i ) {
                if( h.L[i] != -1 ) cout << h.L[i] << " " << i <<
                    endl;
            }
        }
    }
}

```

1.10.3 minCostflow by dacin21(1)

```

/**
// Push-Relabel implementation of the cost-scaling
algorithm
// Runs in O( <max_flow> * log(V * max_edge_cost)) = O(
V^3 * log(V * C))
// Operates on integers
// Works on regular directed graphs.
// Can't operate on doubles. For this, use bellman
ford/dijkstra method.
//Whether or not it works on undirected or multi-edge
or self-loop graphs is yet to be verified.
// To get the actual flow collect all the edges which
have (f > 0) - this works just like the actual
flow finding in normal max flow algorithms;
// Can't find nonMaxflow value upto K. The code is too
complex for me to add this function. Maybe its not

```

```

possible at all and you can only do it with the
bellmanFord/Dijkstra method.
*/
#include<bits/stdc++.h>
using namespace std;
#define optimize()
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
#define endl '\n'
template<typename flow_t = int, typename cost_t = int>
    //int/long long; int/long long;
struct mcSFlow{
    struct Edge{
        cost_t c;
        flow_t f;
        int to, rev;
        Edge(int _to, cost_t _c, flow_t _f, int
            _rev):c(_c), f(_f), to(_to), rev(_rev){}
    };
    const cost_t INFCOST =
        numeric_limits<cost_t>::max()/2; ///divide by
        slightly bigger number if overflow;
    const cost_t INFFLOW =
        numeric_limits<flow_t>::max()/2; ///divide by
        slightly bigger number if overflow;
    cost_t epsilon;
    int N, S, T;
    vector<vector<Edge> > G;
    vector<unsigned int> isEnqueued, state;
    mcSFlow(int _N):epsilon(0), N(_N), G(_N){}
    void add_edge(int a, int b, cost_t cost, flow_t cap){
        if(a==b){assert(cost>=0); return;}
        cost*=N; // to preserve integer-values
        epsilon = max(epsilon, abs(cost));
        assert(a>=0&&a<N&&b>=0&&b<N);
        G[a].emplace_back(b, cost, cap, G[b].size());
        G[b].emplace_back(a, -cost, 0, G[a].size()-1);
    }
    flow_t calc_max_flow(){ // Dinic max-flow
        vector<flow_t> dist(N), state(N);
        vector<Edge*> path(N);
        auto cmp = [](Edge*a, Edge*b){return a->f < b->f;};
        flow_t addFlow, retflow=0;
        do{
            fill(dist.begin(), dist.end(), -1);
            dist[S]=0;
            auto head = state.begin(), tail = state.end();
            for(*tail++ = S; head!=tail; ++head){
                for(Edge const&e:G[*head]){
                    if(e.f && dist[e.to]==-1){
                        dist[e.to] = dist[*head]+1;
                        *tail++=e.to;
                    }
                }
            }
        }
    }
}

```

```

}
addFlow = 0;
fill(state.begin(), state.end(), 0);
auto top = path.begin();
Edge dummy(S, 0, INFFLOW, -1);
*top++ = &dummy;
while(top != path.begin()){
    int n = (*prev(top))->to;
    if(n==T){
        auto next_top = min_element(path.begin(), top,
            cmp);
        flow_t flow = (*next_top)->f;
        while(--top!=path.begin()){
            Edge &e=*top, &f=G[e.to][e.rev];
            e.f-=flow;
            f.f+=flow;
        }
        addFlow+=flow;
        retflow+=flow;
        top = next_top;
        continue;
    }
    for(int &i=state[n], i_max = G[n].size(), need
        = dist[n+1]; ++i){
        if(i==i_max){
            dist[n]=--i;
            --top;
            break;
        }
        if(dist[G[n][i].to] == need && G[n][i].f){
            *top++ = &G[n][i];
            break;
        }
    }
}
}while(addFlow);
return retflow;
}
vector<flow_t> excess;
vector<cost_t> h;
void push(Edge &e, flow_t amt){
    //cerr << "push: " << G[e.to][e.rev].to << " -> "
    << e.to << " (" << e.f << "/" << e.c << ") : "
    << amt << "\n";
    if(e.f < amt) amt=e.f;
    e.f-=amt;
    excess[e.to]+=amt;
    G[e.to][e.rev].f+=amt;
    excess[G[e.to][e.rev].to]-=amt;
}
void relabel(int vertex){
    cost_t newHeight = -INFCOST;
    for(unsigned int i=0; i<G[vertex].size(); ++i){

```

```

        Edge const&e = G[vertex][i];
        if(e.f && newHeight < h[e.to]-e.c){
            newHeight = h[e.to] - e.c;
            state[vertex] = i;
        }
    }
    h[vertex] = newHeight - epsilon;
}
const int scale=2;
pair<flow_t, cost_t> minCostFlow(int _S, int _T){
    S = _S, T = _T;
    cost_t retCost = 0;
    for(int i=0; i<N; ++i){
        for(Edge &e:G[i]){
            retCost += e.c*(e.f);
        }
    }
    //find feasible flow
    flow_t retFlow = calc_max_flow();
    excess.resize(N); h.resize(N);
    queue<int> q;
    isEnqueued.assign(N, 0); state.assign(N, 0);
    for(; epsilon>epsilon>=scale){
        //refine
        fill(state.begin(), state.end(), 0);
        for(int i=0; i<N; ++i)
            for(auto &e:G[i])
                if(h[i] + e.c - h[e.to] < 0 && e.f) push(e,
                    e.f);
        for(int i=0; i<N; ++i){
            if(excess[i]>0){
                q.push(i);
                isEnqueued[i]=1;
            }
        }
        while(!q.empty()){
            int cur=q.front(); q.pop();
            isEnqueued[cur]=0;
            // discharge
            while(excess[cur]>0){
                if(state[cur] == G[cur].size()){
                    relabel(cur);
                }
                for(unsigned int &i=state[cur], max_i =
                    G[cur].size(); i<max_i; ++i){
                    Edge &e=G[cur][i];
                    if(h[cur] + e.c - h[e.to] < 0){
                        push(e, excess[cur]);
                        if(excess[e.to]>0 && isEnqueued[e.to]==0){
                            q.push(e.to);
                            isEnqueued[e.to]=1;
                        }
                    }
                    if(excess[cur]==0) break;
                }
            }
        }
    }
}

```

```

    }
}
}
if(epsilon>1 && epsilon>>scale==0){
    epsilon = 1<<scale;
}
}
for(int i=0; i<N; ++i){
    for(Edge &e:G[i]){
        retCost -= e.c*(e.f);
    }
}
//cerr << " -> " << retFlow << " / " << retCost <<
    " bzw. " << retCost/2/N << "\n";
return make_pair(retFlow, retCost/2/N);
}
flow_t getFlow(Edge const &e){
    return G[e.to][e.rev].f;
}
};
int main() {
    optimize();
    int T;
    cin >> T;
    for( int test = 1; test <= T; ++test ) {
        int N, M, S, T; // Number of nodes; number of
            edges; source; sink;
        cin >> N >> M >> S >> T;
        mcSFlow<> fl(N+1); //Add long long, long long if
            necessary;
        for( int i = 1; i <= M; ++i ) {
            int u, v, c, w; // node; node; cost; capacity;
            cin >> u >> v >> c >> w;
            fl.add_edge(u, v, c, w); //Directed graph;
        }
        cout << fl.minCostFlow(S, T).first << " " <<
            fl.minCostFlow(S, T).second; //flow; cost;
    }
}

```

1.11 geometry

1.11.1 Hill Climbing algorithm

```

// Be intuitive. This algorithm is all about finding
    the right balance among multiplier, 0.998 and the
    number of iterations;
// multiplier = 1.0 ( 0.1 will also do )
// multiplier *= 0.998 ( 0.999 or 0.995 will also
    probably do )

```

```

/// try to set the number of iterations as high as
possible.

```

1.11.2 closest pair of points

```

//O(nlgn)
//The implementation is for double version.
//modify for int/long long version. take help of
comments in code.
double INF = 1e100;
double EPS = 1e-12;
struct PT {
    PT () {}
    double x, y;
    int id;
    PT(double x_, double y_, int id_) : x(x_), y(y_),
        id(id_) {}
};
struct cmp_x {
    bool operator()(const PT &a, const PT &b) const {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
};
struct cmp_y {
    bool operator()(const PT &a, const PT &b) const {
        return a.y < b.y;
    }
};
vector<PT> points; //contains input points;
vector<PT> temp_points;
double mindist; //change double to int/long long as
per necessity
pair<int, int> best_pair; //contain best pair of
points' indexes
void upd_ans(const PT &a, const PT &b) {
    double dist = sqrt((a.x - b.x)*(a.x - b.x) + (a.y -
        b.y)*(a.y - b.y)); //remove sqrt() and store in
int/long long variable as necessary
    if (dist + EPS < mindist) { // remove EPS if working
with int/long long
        mindist = dist;
        best_pair = {a.id, b.id};
    }
}
void ClosestPair(int l, int r) {
    if (r - l <= 3) {
        for (int i = l; i < r; ++i) {
            for (int j = i + 1; j < r; ++j) {
                upd_ans(points[i], points[j]);
            }
        }
    }
}

```

```

        sort(points.begin() + l, points.begin() + r,
            cmp_y());
        return;
    }
    int mid = (l + r) >> 1;
    double midx = points[mid].x; //take int/long long as
necessary
    ClosestPair(l, mid);
    ClosestPair(mid, r);
    merge(points.begin() + l, points.begin() + mid,
        points.begin() + mid, points.begin() + r,
        temp_points.begin(), cmp_y());
    copy(temp_points.begin(), temp_points.begin() + r -
        l, points.begin() + l);
    int tsz = 0;
    for (int i = l; i < r; ++i) {
        //take if((points[i].x - midx)*(points[i].x -
midx) < mindist) as necessary
        if (abs(points[i].x - midx)*EPS < mindist) {
            //take j >= 0 &&
            (points[i].y-temp_points[j].y)*(points[i].y-temp_points[j].y)
            < mindist as necessary
            for (int j = tsz - 1; j >= 0 && points[i].y -
temp_points[j].y + EPS < mindist; --j)
                upd_ans(points[i], temp_points[j]);
            temp_points[tsz++] = points[i];
        }
    }
}
int main() {
    int n; //number of points
    cin >> n;
    points.clear();
    for (int i = 0; i < n; ++i) {
        double x, y;
        cin >> x >> y;
        points.PB( PT(x, y, i) );
    }
    temp_points.clear();
    temp_points.resize((int)points.size());
    sort(points.begin(), points.end(), cmp_x());
    //points vector contains the input points
    //Double version:
    mindist = INF;
    ClosestPair(0, (int)points.size());
    double ans = mindist;
    /** //int/long long version:
    mindist = infLL;
    ClosestPair(0, (int)points.size());
    double ans = sqrt( (double)mindist );
    **/
    cout << ans << endl; // minimum distance
}

```

```

    cout << best_pair.F << " " << best_pair.S << endl;
    //optimal pair of points' indexes
}

```

1.11.3 comparison of doubles

1. `bool equalTo (double a, double b){ if (abs (a - b) <= eps) return true; else return false; }`
2. `bool notEqual (double a, double b){if (abs (a - b) > eps) return true; else return false; }`
3. `bool lessThan (double a, double b){ if (a + eps < b) return true; else return false; }`
4. `bool lessThanEqual (double a, double b){if (a < b + eps) return true; else return false;}`
5. `bool greaterThan (double a, double b){if (a > b + eps) return true;else return false;}`
6. `bool greaterThanEqual (double a, double b){if (a + eps > b) return true;else return false;}`

1.11.4 convex hull 2D

```

//Complexity : O(NlogN)
// returns points of convex hull in CW direction.
double EPS = 1e-12;
struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x,
        y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x,
        y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c
        ); }
    PT operator / (double c) const { return PT(x/c, y/c
        ); }
    bool operator <(const PT &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};
ostream &operator<<(ostream &os, const PT &p) {
    os << "(" << p.x << ", " << p.y << ")";
}
// checks if a-b-c is CW or not.
bool isPointsCW(PT a, PT b, PT c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y)+EPS
        < 0;
}

```

```

}
// checks if a-b-c is CCW or not.
bool isPointsCCW(PT a, PT b, PT c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) >
        EPS;
}
// checks if a-b-c is collinear or not.
bool isPointsCollinear(PT a, PT b, PT c) {
    return abs(a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y))
        <= EPS;
}
void convex_hull(vector<PT>& a) { ///^ == upper hull, U
    == lower hull
    int a_sz = a.size();
    if (a_sz == 1) return;
    sort(a.begin(), a.end());
    PT p1 = a[0], p2 = a.back();
    vector<PT> up, down; // up will store upper
        hull/lower envelope, down will store lower
        hull/upper envelope
    up.push_back(p1);
    down.push_back(p1);
    int up_sz = 1, down_sz = 1;
    for (int i = 1; i < a_sz; i++) {
        if (i == a_sz - 1 || isPointsCW(p1, a[i], p2)) {
            //include all the collinear points of the boundary
            //instead of only the outer two points:-
            //while (up_sz >= 2 && (!isPointsCW(up[up_sz-2],
            //up[up_sz-1], a[i]) &&
            //!isPointsCollinear(up[up_sz-2], up[up_sz-1],
            //a[i])))
            while (up_sz >= 2 && !isPointsCW(up[up_sz-2],
                up[up_sz-1], a[i])) {
                up.pop_back();
                --up_sz;
            }
            up.push_back(a[i]);
            ++up_sz;
        }
        if (i == a_sz - 1 || isPointsCCW(p1, a[i], p2)) {
            //include all the collinear points of the boundary
            //instead of only the outer two points:-
            //while (down_sz >= 2 && (!isPointsCCW(down[down_sz-2],
            //down[down_sz-1], a[i]) &&
            //!isPointsCollinear(down[down_sz-2],
            //down[down_sz-1], a[i])))
            while (down_sz >= 2 &&
                !isPointsCCW(down[down_sz-2],
                down[down_sz-1], a[i])) {
                down.pop_back();
                --down_sz;
            }
            down.push_back(a[i]);

```

```

        ++down_sz;
    }
}
// up has upper hull/lower envelope
// down has lower hull/upper envelope
// a has the overall convex hull
a.clear();
for (int i = 0; i < up_sz; i++) a.push_back(up[i]);
for (int i = down_sz - 2; i > 0; i--)
    a.push_back(down[i]);
}

```

1.11.5 geo routine

```

#include <bits/stdc++.h>
using namespace std;
#define optimize()
    ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
#define endl '\n'
typedef long long ll;
typedef vector<int> vi;
typedef vector<ll> vl;
typedef pair<int,int> pii;
typedef pair<ll,ll> pll;
typedef vector<pii> vii;
typedef vector<pll> vll;
typedef long double ld;
#define F first
#define S second
#define MP make_pair
#define PB push_back
double INF = 1e100;
double EPS = 1e-12;
struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x,
        y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x,
        y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c
        ); }
    PT operator / (double c) const { return PT(x/c, y/c
        ); }
    bool operator <(const PT &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
    bool operator ==(const PT &p) const {
        return (abs(x-p.x) <= EPS && abs(y-p.y) <= EPS);
    }
}

```

```

}
};
double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
    os << "(" << p.x << ", " << p.y << ")";
}
// checks if a-b-c is CW or not.
bool isPointsCW(PT a, PT b, PT c) {
    return
        a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y)+EPS
        < 0;
}
// checks if a-b-c is CCW or not.
bool isPointsCCW(PT a, PT b, PT c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) >
        EPS;
}
// checks if a-b-c is collinear or not.
bool isPointsCollinear(PT a, PT b, PT c) {
    return
        abs(a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y))
        <= EPS;
}
// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p) { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) { // rotate a point CCW t
    degrees around the origin
    return PT(p.x*cos(t)-p.y*sin(t),
        p.x*sin(t)+p.y*cos(t));
}
// rotate a point p CCW around another point q by angle
t(radian)
PT RotateCCW_around_point(PT p, PT q, double t) {
    return PT((p.x-q.x)*cos(t)-(p.y-q.y)*sin(t)+q.x,
        (p.x-q.x)*sin(t)+(p.y-q.y)*cos(t)+q.y);
}
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b-a,b-a);
    if (abs(r) < EPS) return a;
    r = dot(c-a, b-a)/r;
    if (r+EPS < 0) return a;
    if (r > 1+EPS) return b;
    return a + (b-a)*r;
}

```



```

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}
// compute distance between point (x,y,z) and plane
ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                           double a, double b, double c,
                           double d) {
    return abs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}
// determine if lines from a to b and c to d are
parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
    return abs(cross(b-a, c-d)) < EPS;
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
    && abs(cross(a-b, a-c)) < EPS
    && abs(cross(c-d, c-a)) < EPS;
}
// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
            dist2(b, c) < EPS || dist2(b, d) < EPS) return
            true;
        if (dot(c-a, c-b) > EPS && dot(d-a, d-b) > EPS &&
            dot(c-b, d-b) > EPS)
            return false;
        return true;
    }
    if (cross(d-a, b-a) * cross(c-a, b-a) > EPS) return
        false;
    if (cross(a-c, d-c) * cross(b-c, d-c) > EPS) return
        false;
    return true;
}
// compute intersection of line passing through a and b
// with line passing through c and d, assuming that
// unique
// intersection exists; for segment intersection, check
// if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b=b-a; d=c-d; c=c-a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
    return a + b*cross(c, d)/cross(b, d);
}
// shift the straight line passing through points a and
// b
// by distance Dist.

```

```

// If Dist is negative the line is shifted rightwards
// or upwards.
// If Dist is positive the line is shifted leftwards or
// downwards.
// The new line passes through points c and d
pair<PT,PT> ShiftLineByDist(PT a, PT b, double Dist) {
    double r = sqrt(dist2(a, b));
    double delx = (Dist*(a.y-b.y))/r;
    double dely = (Dist*(b.x-a.x))/r;
    PT c = PT(a.x+delx, a.y+dely);
    PT d = PT(b.x+delx, b.y+dely);
    return MP(c, d);
}
// compute intersection of circle centered at a with
// radius r
// with circle centered at b with radius R
// *** The centers of the two circles must not
// coincide. i.e (a != b)
vector<PT> CircleCircleIntersection(PT a, PT b, double
r, double R) {
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r+R || d+min(r, R) < max(r, R)) return ret;
    double x = (d*d-R*R+r*r)/(2*d);
    double y = sqrt(r*r-x*x);
    PT v = (b-a)/d;
    ret.push_back(a+v*x + RotateCCW90(v)*y);
    if (y > 0)
        ret.push_back(a+v*x - RotateCCW90(v)*y);
    return ret;
}
// This code computes the area or centroid of a
// (possibly nonconvex)
// polygon, assuming that the coordinates are listed in
// a clockwise or
// counterclockwise fashion. Note that the centroid is
// often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area / 2.0;
}
double ComputeArea(const vector<PT> &p) {
    return abs(ComputeSignedArea(p));
}
PT ComputeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 * ComputeSignedArea(p);
    for (int i = 0; i < p.size(); i++){

```

```

        int j = (i+1) % p.size();
        c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}
// angle from p2->p1 to p2->p3, returns -PI to PI (but
// you may choose to return the absolute value only
// which is "more practical").
double angle(PT p1, PT p2, PT p3) {
    PT va = p1-p2, vb=p3-p2;
    double x,y;
    x=dot(va,vb);
    y=cross(va,vb);
    return(atan2(y,x)); /// return abs( atan2(y,x) ) to
    get the absolute value of angle. (tested)
}
double angle_to_radian( double theta ) {
    return ((M_PI/180.0)*theta);
}
double radian_to_angle( double x ) {
    return ((180.0/M_PI)*x);
}
int main() {
    // expected: (-5,2)
    cerr << RotateCCW90(PT(2,5)) << endl;
    // expected: (5,-2)
    cerr << RotateCW90(PT(2,5)) << endl;
    // expected: (-5,2)
    cerr << RotateCCW(PT(2,5),M_PI/2) << endl;
    // expected: (5,2)
    cerr << ProjectPointLine(PT(-5,-2), PT(10,4),
        PT(3,7)) << endl;
    // expected: (5,2) (7.5,3) (2.5,1)
    cerr << ProjectPointSegment(PT(-5,-2), PT(10,4),
        PT(3,7)) << " "
        << ProjectPointSegment(PT(7.5,3), PT(10,4),
        PT(3,7)) << " "
        << ProjectPointSegment(PT(-5,-2), PT(2.5,1),
        PT(3,7)) << endl;
    // expected: 6.78903
    cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;
    // expected: 1 0 1
    cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1),
        PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(2,0),
        PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(5,9),
        PT(7,13)) << endl;
    // expected: 0 0 1
    cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1),
        PT(4,5)) << " "
        << LinesCollinear(PT(1,1), PT(3,5), PT(2,0),
        PT(4,5)) << " "

```

```

    << LinesCollinear(PT(1,1), PT(3,5), PT(5,9),
        PT(7,13)) << endl;
// expected: (1,2)
cerr << ComputeLineIntersection(PT(0,0), PT(2,4),
    PT(3,1), PT(-1,3)) << endl;
// area should be 5.0
// centroid should be (1.16666666, 1.16666666)
PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
vector<PT> p(pa, pa+4);
PT c = ComputeCentroid(p);
cerr << "Area: " << ComputeArea(p) << endl;
cerr << "Centroid: " << c << endl;
// expected: 0
cerr << isPointsCCW( PT(5, 6), PT(10, 10), PT(11, 5)
    ) << endl;
// expected: 1
cerr << isPointsCCW( PT(5, 6), PT(10, 2), PT(11, 5) )
    << endl;
// expected: 1
cerr << isPointsCW( PT(5, 6), PT(10, 10), PT(11, 5) )
    << endl;
// expected: 0
cerr << isPointsCW( PT(5, 6), PT(10, 2), PT(11, 5) )
    << endl;
// expected: 0
cerr << isPointsCollinear( PT(5, 6), PT(10, 2),
    PT(11, 5) ) << endl;
// expected: 1
cerr << isPointsCollinear( PT(5, 6), PT(10, 6),
    PT(11, 6) ) << endl;
// expected: (-0.437602,12.6564) (2.5624,14.6564)
cerr << ShiftLineByDist( PT(4, 6), PT(7, 8), 8 ).F <<
    " " << ShiftLineByDist( PT(4, 6), PT(7, 8), 8
    ).S << endl;
// expected: (8.4376,-0.656402) (11.4376,1.3436)
cerr << ShiftLineByDist( PT(4, 6), PT(7, 8), -8 ).F
    << " " << ShiftLineByDist( PT(4, 6), PT(7, 8),
    -8 ).S << endl;
}

```

1.11.6 half plane intersection Radewoosh style (OFFLINE)

```

// OFFLINE
// Complexity: O(NlgN)
// very easy concept and implementation
double INF = 1e100;
double EPS = 1e-12;
struct PT {
    double x, y;

```

```

    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x,
        y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x,
        y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c
        ); }
    PT operator / (double c) const { return PT(x/c, y/c
        ); }
    bool operator <(const PT &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};
ostream &operator<<(ostream &os, const PT &p) {
    os << "(" << p.x << "," << p.y << ")";
}
int steps = 600;
vector<PT> lower_hull, upper_hull;
int lower_hull_sz, upper_hull_sz;
bool leBorder = 0, riBorder = 0;

double func( double xx, double val ) {
    double ans1 = INF, ans2 = -INF, ans;
    for( int i = 0; i < lower_hull_sz-1; ++i ) {
        if( leBorder && (i == 0) ) continue;
        PT a = lower_hull[i], b = lower_hull[i+1]; //
            straight line passes through points a and b
        double m = (a.y-b.y)/(a.x-b.x); // slope of the
            straight line; if the TL is strict, then
            better precalculate all the slopes and store
            them beforehand
        double c = a.y - a.x*m; // intercept of the
            straight line; if the TL is strict, then
            better precalculate all the intercepts and
            store them beforehand
        double aa = m*xx;
        double bb = c;
        double cc = aa+bb;
        ans1 = min( ans1, cc );
    }
    for( int i = 0; i < upper_hull_sz-1; ++i ) {
        if( riBorder && (i == upper_hull_sz-2) ) continue;
        PT a = upper_hull[i], b = upper_hull[i+1]; //
            straight line passes through points a and b
        double m = (a.y-b.y)/(a.x-b.x); // slope of the
            straight line; if the TL is strict, then
            better precalculate all the slopes and store
            them beforehand
        double c = a.y - a.x*m; // intercept of the
            straight line; if the TL is strict, then
            better precalculate all the intercepts and

```

```

            store them beforehand
        double aa = m*xx;
        double bb = c;
        double cc = aa+bb;
        ans2 = max( ans2, cc );
    }
    ans = ans1-ans2;
    return ans;
}
bool Ternary_Search(double val) {
    double lo = -INF, hi = INF, mid1, mid2;
    leBorder = 0, riBorder = 0;
    if( lower_hull[0].x == lower_hull[1].x ) lo =
        lower_hull[0].x+val, leBorder = 1;
    if( upper_hull[upper_hull_sz-2].x ==
        upper_hull[upper_hull_sz-1].x ) hi =
        upper_hull[upper_hull_sz-1].x-val, riBorder = 1;
    if( lo > hi ) return 0;
    for( int i = 0; i < steps; ++i ) {
        mid1 = (lo*2.0 + hi)/3.0;
        mid2 = (lo + 2.0*hi)/3.0;
        double ff1 = func(mid1, val);
        double ff2 = func(mid2, val);
        if( ff1 >= 0 || ff2 >= 0 ) return 1;
        if( ff1 > ff2 ) hi = mid2;
        else lo = mid1;
    }
    if( func(lo, val) >= 0 ) return 1;
    return 0;
}

```

1.12 graph

1.12.1 FloydWarshall

```

// order of nodes: k->i->j

```

1.12.2 articulation bridges

```

//complexity: O(E+V)
//nodes are 1-based indexed.
int n; // number of nodes
vector<vector<int>> adjlist; // adjacency list of graph
vector<bool> visited;
vector<int> tin, low;
int timer;
void dfs_art(int u, int par = -1) {
    visited[u] = true;

```

```

tin[u] = low[u] = timer++;
for (auto to : adjlist[u]) {
    if (to == par) continue;
    if (visited[to]) {
        low[u] = min(low[u], tin[to]);
    }
    else {
        dfs_art(to, u);
        low[u] = min(low[u], low[to]);
        if (low[to] > tin[u]) {
            IS_BRIDGE(u, to);
        }
    }
}
}
}
void find_bridges() {
    timer = 0;
    visited.clear(); tin.clear(); low.clear();
    visited.assign(n+1, false);
    tin.assign(n+1, -1);
    low.assign(n+1, -1);
    for (int i = 1; i <= n; ++i) {
        if (!visited[i]) dfs_art(i);
    }
}

```

1.12.3 centroid decomposition

```

const int MAXN = 1e5+5; /// Maximum no. of nodes;
int n; /// Number of nodes;
vector<int> adjlist[MAXN];
int sub_cd[MAXN]; /// sub_cd[u] = subtree size of node
u after each decomposing;
bool done_cd[MAXN]; /// done_cd[u] = node u have been
decomposed;
void dfs_sub_cd(int u, int par) {
    sub_cd[u] = 1;
    for( auto v: adjlist[u] ) {
        if( done_cd[v] || v == par ) continue;
        dfs_sub_cd(v, u);
        sub_cd[u] += sub_cd[v];
    }
}
int dfs_find_centroid( int u, int par, int sz ) {
    for( auto v: adjlist[u] ) {
        if( !done_cd[v] && v != par && sub_cd[v] > sz ) {
            return dfs_find_centroid( v, u, sz );
        }
    }
    return u;
}

```

```

void dfs_decompose( int u ) {
    dfs_sub_cd(u, -1);
    int centroid = dfs_find_centroid(u, -1, sub_cd[u]/2);
    done_cd[centroid] = 1;
    for( auto v : adjlist[centroid] ) {
        if( !done_cd[v] ) dfs_decompose(v);
    }
}

```

1.12.4 djikstra

```

///O((V+E)logV)
///Works on negative edges(a bit slow) but not negative
cycle
const int MAXN = 1e5+5;
vi adjlist[MAXN];
int dist[MAXN];
void djikstra( int source ) {
    for( int i = 0; i < MAXN; ++i ) dist[i] = inf;
    dist[source] = 0;
    priority_queue< pii, vector<pii>, greater<pii> > pq;
    pq.push(MP(0, source));
    while (!pq.empty()) {
        pii Front = pq.top(); pq.pop();
        int d = Front.first, u = Front.second;
        if (d > dist[u]) continue;
        for (int j = 0; j < adjlist[u].size(); j++) {
            pii v = adjlist[u][j];
            if (dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(MP(dist[v.first], v.first));
            }
        }
    }
}

```

1.12.5 dsu on trees (contest version)

```

int cnt[MAXN];
bool big[MAXN];
void add(int u, int par, int x){
    cnt[ col[u] ] += x;
    for(auto v: adjlist[u])
        if(v != par && !big[v]) add(v, u, x);
}
void dfs(int u, int par, bool keep){
    int mx_val = -1, bigChild = -1;
    for(auto v : adjlist[u])

```

```

        if(v != par && sz[v] > mx_val) mx_val = sz[v],
            bigChild = v;
    for(auto v : adjlist[u])
        if(v != par && v != bigChild)
            dfs(v, u, 0); // run a dfs on small childs and
                            clear them from cnt
    if(bigChild != -1)
        dfs(bigChild, u, 1), big[bigChild] = 1; // bigChild
        marked as big and not cleared from cnt
    add(u, par, 1);
    //now cnt[c] is the number of vertices in subtree of
    vertex u that has color c. You can answer the
    queries easily.
    if(bigChild != -1) big[bigChild] = 0;
    if(keep == 0) add(u, par, -1);
}

```

But why it is $O(n \log n)$? You know that why dsu has $O(q \log n)$ time (for q queries); the code uses the same method. Merge smaller to greater.

If you have heard heavy-light decomposition you will see that function add will go light edges only, because of this, code works in $O(n \log n)$ time. Any problems of this type can be solved with same dfs function and just differs in add function.

1.12.6 euler circuit and path(2 in one)

```

///The implementation for euler circuit and euler path
is exactly same.
///The implementation for directed and undirected graph
is exactly same.

```

- ```

/**
1. Path traversing all the edges just once starting
from one node and ending at that node is euler
circuit.
2. If the path does not end at the source node then
that path is euler path.

```

- ```

**/
1. euler circuit/euler path for undirected graph
/**
1. The graph must be a single component. So, you have
to check it.
2. All the nodes should have even degree. So, check
it. But if just 2 nodes have odd degree then
there will an eulerian path only. Choose any of
these 2 nodes as source node in that case.
3. If node u has a self loop then deg[u] += 2; i.e.
the degree of node u is increased by 2.
**/
2. Euler circuit/euler path for directed graph
/**

```

1. The graph -if converted to undirected must be a single component. So, you have to check it.
2. in-degree and out-degree of all nodes should be equal. So, check it. But if just 2 nodes have unequal in-degree and out-degree then there will an eulerian path only if and only if one these node(source node to be exact) have out-degree==1+in-degree and the other have in degree==1+out-degree. Choose any of these 2 nodes as source node in that case.
3. If node u has a self loop then in-degree[u]++; and out-degree[u]++;

```

**/
vector<pii> gvec[10005]; //adjacency list; gvec[u]
    contains pairs (v, id) where v is node and id is
    the edge no.
bool bad[10005]; //size of this array is the number of
    edges
vector<int> circuit; //contains the path
void eularcircuit(int src) {
    stack<int> curr_path;
    circuit.clear();
    if(gvec[src].empty())return;
    int curr_v = src;
    while (1) {
        if (!gvec[curr_v].empty()) {
            auto it=gvec[curr_v].back();
            int next_v = it.F, id = it.S;
            gvec[curr_v].pop_back();
            if( bad[id] ) continue;
            bad[id] = 1;
            curr_path.push(curr_v);
            curr_v = next_v;
        }
        else {
            circuit.push_back(curr_v);
            if(curr_path.empty()) break;
            curr_v = curr_path.top();
            curr_path.pop();
        }
    }
    reverse(circuit.begin(),circuit.end());
}
int main() {
    int n, m; // no. of nodes, no. of edges;
    cin >> n >> m;
    for( int i = 0; i < 10005; ++i ) gvec[i].clear();
    memset( bad, 0, sizeof(bad) );
    circuit.clear();
    for( int i = 0; i < m; ++i ) {
        int u, v;
        cin >> u >> v;
        gvec[u].PB( MP(v, i) ); //Directed graph

```

```

    /** Undirected graph
        gvec[u].PB( MP(v, i) );
        gvec[v].PB( MP(u, i) );
    **/
}
eularcircuit( source );
}

```

1.12.7 hld

```

//HLD path query + Euler tour(pre-order traversal)
    subtree query

const int MAXN = 1e5+5; // max no. of nodes;
int n; //number of nodes;

vector<int> adjlist[MAXN];
int sub[MAXN]; //sub[u] = size of subtree u;
int head[MAXN]; //head[u] = head of node u's chain;
int par[MAXN]; //par[u] = parent of node u; parent of
    root node is -1;
int depth[MAXN]; //depth[u] = depth of node u;

int tin[MAXN];
int tout[MAXN];
int timer;

/**
No need to clear at the beginning of a test case: sub,
    head, par, depth, tin, tout;
**/

/**
1. Position of node u in segment tree is: tin[u];
2. Subtree query(u): [tin[u], tout[u]]
3. Path query(u, v): [tin[u], tin[v]] (here u and v are
    of the same chain and depth[u] <= depth[v])
4. u--v is a heavy edge if and only if depth[u] ==
    depth[v]-1 and adjlist[u][0] = v; Thus u and v are
    of the same chain;
**/

int arr[MAXN]; // contains the values for the segment
    tree
int tree[MAXN * 4];

void init_hld( int node, int l, int r ) {
    if( l == r ) {
        tree[node] = arr[l];
        return;
    }

```

```

    int mid = (l + r)/2;
    init_hld( 2*node, l, mid );
    init_hld( 2*node+1, mid+1, r );
    tree[node] = max( tree[2*node], tree[2*node+1] );
}

void update_hld( int node, int l, int r, int b, int x )
    { //point update
    if( l == r ) {
        tree[node] = x;
        return;
    }
    int mid = (l + r)/2;
    if( mid >= b ) update_hld( 2*node, l, mid, b, x );
    else update_hld( 2*node+1, mid+1, r, b, x );
    tree[node] = max( tree[2*node], tree[2*node+1] );
}

int query_hld( int node, int l, int r, int b, int e ) {
    if( l > e || r < b ) return -inf;
    if( l >= b && r <= e ) return tree[node];
    int mid = (l + r)/2;
    return max( query_hld( 2*node, l, mid, b, e ),
        query_hld( 2*node+1, mid+1, r, b, e ) );
}

void dfs_sub( int u, int p, int dep ) {
    sub[u] = 1;
    par[u] = p;
    depth[u] = dep;
    int cur = 0;
    for( auto &v: adjlist[u] ) {
        if( v == p ) continue;
        dfs_sub(v, u, dep+1);
        sub[u] += sub[v];
        if(sub[v] > cur) {
            cur = sub[v];
            swap(v, adjlist[u][0]);
        }
    }
}

void dfs_hld( int u, int p ) {
    tin[u] = timer++;
    for( auto v: adjlist[u] ) {
        if( v == p ) continue;
        head[v] = (v == adjlist[u][0] ? head[u] : v);
        dfs_hld(v, u);
    }
    tout[u] = timer-1;
}

void preprocess(int root) {
    timer = 1;
    head[root] = root;
}

```

```

    dfs_sub(root, -1, 0);
    dfs_hld(root, -1);
}

int ultimate_query_hld(int u, int v) {
    int ans = -inf;
    while(head[u] != head[v]) {
        if(depth[head[u]] > depth[head[v]]) swap(u, v);
        ans = max(ans, query_hld(1, 1, n, tin[head[v]],
            tin[v]));
        v = par[head[v]];
    }
    if(depth[u] > depth[v]) swap(u, v);
    ans = max(ans, query_hld(1, 1, n, tin[u], tin[v]));
    return ans;
}

void ultimate_update_hld(int u, int v, int x) {
    while(head[u] != head[v]) {
        if(depth[head[u]] > depth[head[v]]) swap(u, v);
        update_hld(1, 1, n, tin[head[v]], tin[v], x);
        v = par[head[v]];
    }
    if(depth[u] > depth[v]) swap(u, v);
    update_hld(1, 1, n, tin[u], tin[v], x);
}

int lca(int u, int v) {
    while(head[u] != head[v]) {
        if(depth[head[u]] > depth[head[v]]) swap(u, v);
        v = par[head[v]];
    }
    if(depth[u] > depth[v]) swap(u, v);
    return u;
}

```

1.12.8 lca

```

int n, l;
vector<vector<int>> adjlist;
int timer;
vector<int> tin, tout;
vector<vector<int>> up;
vector<int> depth;
void dfs_lca(int u, int par, int dep) {
    depth[u] = dep;
    tin[u] = ++timer;
    up[u][0] = par;
    for (int i = 1; i <= l; ++i) up[u][i] =
        up[up[u][i-1]][i-1];
    for (auto v : adjlist[u]) {

```

```

        if (v != par) dfs_lca(v, u, dep+1);
    }
    tout[u] = ++timer;
}

bool is_ancestor(int u, int v) { //is u an ancestor of v?
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v) {
    if (is_ancestor(u, v)) return u;
    if (is_ancestor(v, u)) return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v)) u = up[u][i];
    }
    return up[u][0];
}

int kth_ancestor(int u, int k) { //kth ancestor of u
    for (int i = l; i >= 0; --i)
        if ((1LL << i) & k) u = up[u][i];
    return u;
}

int get_dist(int u, int v) {
    return depth[u] + depth[v] - 2 * depth[lca(u, v)];
}

void preprocess(int root) {
    tin.resize(n+1);
    tout.resize(n+1);
    depth.resize(n+1);
    timer = 0;
    l = ceil(log2(n+1));
    up.assign(n+1, vector<int>(l + 1));
    dfs_lca(root, root, 0);
}

```

1.12.9 topological sort (tarjan)

```

//complexity: O(N)
vi adjlist[10005];
bool vis[10005];
vector<int> toposort;
void dfs_topo(int u) {
    vis[u] = 1;
    for (auto v : adjlist[u]) {
        if (!vis[v]) dfs_topo(v);
    }
    toposort.pb(u);
}

int main() {
    for (int i = 1; i <= n; ++i) {
        if (!vis[i]) dfs_topo(i);
    }
}

```

```

reverse(toposort.begin(), toposort.end());
}

```

1.12.10 virtual tree

///Complexity for each query: $O(3 * k \log k) \sim O(k \log k)$; k is the number of special nodes in each query;

/**
-Before approaching a virtual tree problem you must be able to understand this code!!!! This code is very easy. This code can be revised in 5 minutes(worst case maybe?)
-But no need to understand the logic of why the code is written that way.

*/
/**
Brief explanation of the code

For each query:

1. sort the nodes according to their dfs starting time
2. find the lca of the adjacent nodes and insert them to the same container
3. sort the nodes again according to their dfs starting time
4. edge banao

/**
-if two nodes u and v belongs to an auxiliary tree then lca(u,v) also belongs to that auxiliary tree.
-for each query: for k special nodes there will be highest $2*k-1$ nodes in the auxiliary tree.

```

const int mx = 1e5+5; //max no. of nodes;

//*****LCA code needed here*****

bool cmp(const int &lhs, const int &rhs) {
    return tin[lhs] < tin[rhs]; // tin is the starting dfs time of a node
}

vi aux[mx]; //auxiliary tree

```

```

int main() {
    /**
    Problem:
    You will be given a tree.

```

```

You will be given many queries. In each query you
will be given k special nodes. Now build an
auxiliary tree out of these k special nodes.
Constraints:
Summation of k over all queries does not exceed 10^5
**/
int q; ///queries;
cin >> q;
while(q--){
    int k; ///how many special nodes?
    cin >> k;
    vector<int> special; ///this contains the special
        nodes;
    for( int i = 0; i < k; ++i ) {
        int node;
        cin >> node;
        special.PB(node);
    }

    /**Node selection for the auxiliary tree - start**/
    sort( special.begin(), special.end(), cmp );
    ///sort the nodes according to their starting
    dfs time;
    special.erase( unique(special.begin(),
        special.end()), special.end() );
    for( int i = special.size()-1; i > 0; --i ) {
        special.PB( lca( special[i-1], special[i] ) );
    }
    sort( special.begin(), special.end(), cmp );
    ///sort the nodes according to their starting
    dfs time;
    special.erase( unique(special.begin(),
        special.end()), special.end() );
    /**Node selection for the auxiliary tree - end**/

    /**Edge selection for the auxiliary tree - start**/
    for( auto node : special ) {
        aux[node].clear(); ///clearing the auxiliary tree
        graph
    }
    stack<int> st;
    st.push(special[0]);
    for( int i = 1; i < special.size(); ++i ) {
        while( !is_ancestor(st.top(), special[i]) )
            st.pop();
        int u = st.top(), v = special[i];
        aux[u].PB(v);
        aux[v].PB(u);
        st.push(special[i]);
    }
    /**Edge selection for the auxiliary tree - end**/

    /**Thus aux contains the auxiliary tree graph**/

```

```

}
}

```

1.13 graph or integer sequence hashing

1.13.1 graph hashing using modulo and prime(Xellos approved)

```

///Graph hashing is not the same as string hashing;
///Be intuitive;
///No idea how this works. But it works;

/**
Q) How to hash the path u-v-w?
A) hash value of the path =
    ((po1[u]+po1[v]+po1[w])%MOD1,
    (po2[u]+po2[v]+po2[w])%MOD2)
**/
const int MAXN = 1e5+5; ///MAX value of nodes;
int n; ///no. of nodes;
int base1 = 131, base2 = 137; ///fairly big prime
    number; prime numbers strictly greater than max
    no. of nodes may also do;
ll po1[MAXN], po2[MAXN]; /// po1[u] and po2[u] stores
    the double hash value of node u;
#define MOD1 1000000007
#define MOD2 1000000009
int main() {
    po1[0] = po2[0] = 1;
    for( int i = 1; i <= n; ++i ) {
        po1[i] = po1[i-1]*base1;
        po1[i] %= MOD1;

        po2[i] = po2[i-1]*base2;
        po2[i] %= MOD2;
    }
}

```

1.13.2 graph hashing using xor(Xellos approved)

```

///Graph hashing is not the same as string hashing;
///Be intuitive;
///No idea how this works. But it works;

/**
Q) How to hash the path u-v-w?
A) There are two ways-
    1. hash value of the path = ((x1[u]^x1[v]^x1[w]),
        (x2[u]^x2[v]^x2[w]))
    2. hash value of the path = ((x1[u]^x1[v]^x1[w]),
        (x2[u]^x2[v]^x2[w]))

Way no-1 can be used any time, but way no-2 can be
used if and only if the path has no repetitive
nodes;
**/
mt19937
rng((unsigned)chrono::system_clock::now().time_since_epoch
const int MAXN = 1e5+5; ///MAX value of nodes;
int n; /// No. of nodes;
ll x1[MAXN], x2[MAXN]; /// x1[u] and x2[u]
    stores the double hash value of node u;
int main() {
    for( int i = 1; i <= n; ++i ) {
        x1[i] = rng(), x2[i] = rng();
    }
}

```

```

/**
Q) How to hash the path u-v-w?
A) There are two ways-
    1. hash value of the path = ((x1[u]^x1[v]^x1[w]),
        (x2[u]^x2[v]^x2[w]))
    2. hash value of the path = ((x1[u]^x1[v]^x1[w]),
        (x2[u]^x2[v]^x2[w]))

Way no-1 can be used any time, but way no-2 can be
used if and only if the path has no repetitive
nodes;
**/
mt19937
rng((unsigned)chrono::system_clock::now().time_since_epoch
const int MAXN = 1e5+5; ///MAX value of nodes;
int n; /// No. of nodes;
ll x1[MAXN], x2[MAXN]; /// x1[u] and x2[u]
    stores the double hash value of node u;
int main() {
    for( int i = 1; i <= n; ++i ) {
        x1[i] = rng(), x2[i] = rng();
    }
}

```

1.14 important C++ features

```

/// 1. Unique vector
sort(vec.begin(), vec.end());
vec.erase( unique( vec.begin(), vec.end() ), vec.end()
);

/// 2. bit manipulation
inline bool checkBit(ll n, int i) { return n&(1LL<<i); }
inline ll setBit(ll n, int i) { return n|(1LL<<i); }
inline ll resetBit(ll n, int i) { return n&~(1LL<<i); }
}

///3. dir array
int dx[] = {0, 0, +1, -1};
int dy[] = {+1, -1, 0, 0};
//int dx[] = {+1, 0, -1, 0, +1, +1, -1, -1};
//int dy[] = {0, +1, 0, -1, +1, -1, +1, -1};

///4. constructor overwriting
struct Vertex {
    int next[K];
    bool leaf = false;
    int p = -1;
    char pch;
    int link = -1;
}

```



```

int go[K];
Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
    fill(begin(next), end(next), -1);
    fill(begin(go), end(go), -1);
}
};

//5. substring copy(Very fast. faster than hand-made
    custom. can avoid TLE)
string s;
s.substr(idx); //copy substring of range [idx,
    s.size()-1]
s.substr(idx, x); //copy substring of range [idx,
    idx+x-1]

//6. priority queue in ascending order
priority_queue<int, vector<int>, greater<int> > pq;

//7. random number generator
mt19937
rng((unsigned)chrono::system_clock::now().time_since_epoch().count());
//mt19937_64 for ll (**use this above main
function**)

shuffle( vec.begin(), vec.end(), rng ); //shuffles a
    vector
int temp = rng(); //generates a random number

//8. make map faster
ump.reserve(1<<10); ump.max_load_factor(0.25);

//9. Architectural improvement. (Use this above header
    file) (Noticeable incase of bitset)
#pragma GCC optimize("Ofast")
#pragma GCC
    target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,avx,avx2,fma")
#pragma GCC optimize("unroll-loops")

//10. count the number of set bits in a number
__builtin_popcount(num); // for integer;
__builtin_popcountll(num); // for long long;

```

1.15 josephus problem

```

// Josephus problem, n people numbered from 1 to n
    stand in a circle.
// Counting starts from 1 and every k'th people dies
// Returns the position of the m'th killed people
// For example if n = 10 and k = 3, then the people
    killed are 3, 6, 9, 2, 7, 1, 8, 5, 10, 4
    respectively

```

```

// 0(n) // tested by me.
int josephus(int n, int k, int m){
    int i;
    for (m = n - m, i = m + 1; i <= n; i++){
        m += k;
        if (m >= i) m %= i;
    }
    return m + 1;
}

// 0(k log(n)) // not tested by me.
long long josephus2(long long n, long long k, long long
    m){ // hash = 583016 (what does hash here mean?
    idk)
    m = n - m;
    if (k <= 1) return n - m;
    long long i = m;
    while (i < n){
        long long r = (i - m + k - 2) / (k - 1);
        if ((i + r) > n) r = n - i;
        if (!r) r = 1;
        i += r;
        m = (m + (r * k)) % i;
    }
    return m + 1;
}

```

1.16 matrix exponentiation

```

struct matrix {
    matrix() {}
    vector<vector<ll>> mat;
    matrix(int n, int m) : mat(n, vector<ll>(m)) {}
};

matrix operator * (matrix a, matrix b) { // pass by
    reference if TLE
    int n = a.mat.size();
    int m = a.mat[0].size();
    assert(b.mat.size() == m);
    int k = b.mat[0].size();
    matrix ans (n, k);
    for (int i=0; i<n; i++){
        for (int j=0; j<k; j++){
            for (int l=0; l<m; l++){
                ans.mat[i][j] = (ans.mat[i][j] +
                    a.mat[i][l]*b.mat[l][j])%MOD;
            }
        }
    }
    return ans;
}

matrix power( matrix b, ll p ) { // pass by reference
    if TLE
    matrix res = I, x = b;

```

```

while(p) {
    if ( p&1LL ) res = res*x;
    x = x*x;
    p >>= 1LL;
}
return res;
}

```

1.17 number theory

1.17.1 Mobius inversion

1.17.2 Tonelli-Shanks(for solving quadratic congruence equation)

```

//solves TIMUS 1132
//can't figure out the time complexity, but its fast

//This function solves quadratic equation modulo prime
    p;
//returns -1 if there is no possible solution. i.e the
    congruence equation is not valid.
//returns an answer(if exists) as integer less than
    prime p.
//This algorithm may work for prime power moduli
    (according to wikipedia, but not tested)

```

```

//solves  $x^2 \equiv a \pmod{p}$ ;

```

```

/**
***Important notes***

```

****For even prime moduli($p = 2$):**

1. There is exactly 1 solution in range $[1, p-1]$. But when range is not concerned then there are infinite solutions. If a solution is x then the other solutions can be defined as $k*p+x$ or $k*p-x$ where k is any integer. you can prove it by showing that $(k*p+x)^2$ and x^2 are congruent modulo p or $(k*p-x)^2$ and x^2 are congruent modulo p .

****For odd prime moduli:**

- 1.If the congruence equation is valid then there are always exactly 2 solutions in range $[1, p-1]$. if a solution is x (returned by the function) then the other solution is $p-x$. (Exception: if congruence equation is valid and $x = 0$ is returned by the function, then $p-x = 0$. So, even though the congruence equation is valid- there are no solutions in range $[1, p-1]$. However, if range is

```

    not concerned then there are infinitely many
    solutions.)
2. But, if range is not concerned then there exists
    infinite solution if the congruence equation is
    valid. If a solution is x then the other solutions
    can be defined as  $k*p+x$  or  $k*p-x$  where k is any
    integer. you can prove it by showing that
     $(k*p+x)^2$  and  $x^2$  are congruent modulo p or
     $(k*p-x)^2$  and  $x^2$  are congruent modulo p.
**/

ll expo( ll b, ll power, ll m ) {
    ll res = 1LL, x = b%m;
    while(power) {
        if ( power&1LL ) res = ( res*x ) % m;
        x = ( x*x ) % m;
        power >>= 1LL;
    }
    return res;
}

//solves  $x*x == a \pmod p$ ; [Here, p is prime. We find a
    possible value of x.]
int solvequadratic(int a, int p) {
    if(p == 2) {
        if(a&1) return 1;
        return 0;
    }
    if(a > p) a = a%p;
    while(a < 0) a = a+p;
    if(a == 0) return 0;
    if(expo(a, (p-1)/2, p) != 1) return -1;

    int n = 0, k = p-1;
    while(k % 2 == 0) k/=2, n++;
    int q = 2;
    while(expo(q, (p-1)/2, p) != (p-1)) q++;
    int t = expo(a, (k+1)/2, p);
    int r = expo(a, k, p);
    while(true) {
        int s = 1;
        int i = 0;
        while(expo(r, s, p) != 1) i+=1, s*=2;
        if(i == 0) return t;
        int e = 1<<(n-i-1);
        int u = expo(q, k*e, p);
        t = ((long long)t*u)%p;
        r = ((long long)r*u*u)%p;
    }
}

```

1.17.3 bigmod

```

ll expo( ll b, ll p, ll m ) {
    ll res = 1LL, x = b%m;
    while(p) {
        if ( p&1LL ) res = ( res*x ) % m;
        x = ( x*x ) % m;
        p >>= 1LL;
    }
    return res;
}

```

1.17.4 factorial factorize

```

//How many times prime p occurs in n! (O(lgn))
long long factorialPrimePower ( long long n, long long
    p ) {
    long long freq = 0;
    long long x = n;
    while ( x ) {
        freq += x / p;
        x = x / p;
    }
    return freq;
}

//prime factorization of n! (O(nlgn))
void factFactorize ( long long n ) {
    for ( int i = 0; i < prime.size() && prime[i] <= n;
        i++ ) {
        ll x = n;
        ll freq = 0;
        while ( x ) {
            freq += x / prime[i];
            x = x / prime[i];
        }
        cout << prime[i] << ' ' << freq << endl;
    }
}

```

1.17.5 mod inverse from 1 to N

```

//mod inverse from 1 to N
//Complexity: O(N)
const int mx = 100005;
ll inv[mx];
void generate_modinv() {
    inv[1] = 1;

```

```

for( int i = 2; i < mx; i++ ) {
    inv[i] = (-(MOD/i) * inv[MOD%i] ) % MOD;
    inv[i] = inv[i] + MOD;
}
}

```

1.17.6 pollard rho

```

/**
Range:  $10^{18}$  (tested), should be okay up to  $2^{63}-1$ 
miller_rabin(n)
    returns 1 if prime, 0 otherwise
    Identifies 70000 18 digit primes in 1 second on Toph
pollard_rho(n):
    If n is prime, returns n
    Otherwise returns a proper divisor of n
    Able to factorize ~120 18 digit semiprimes in 1
        second on Toph
    Able to factorize ~700 15 digit semiprimes in 1
        second on Toph
At the very beginning, call rho::init(); this makes
    miller_rabin and pollard rho functions to work
(Copied from sh_shahin. I slightly modified the
    primeFactorize function to make it a bit faster.
    Changed rand() to rng()).
*/
namespace rho{
    mt19937
        rng(chrono::steady_clock::now().time_since_epoch().count);
    const int MAXP = 1000010;
    ll seq[MAXP];
    int primes[MAXP], spf[MAXP];
    inline ll mod_add(ll x, ll y, ll m){
        return (x += y) < m ? x : x - m;
    }
    inline ll mod_mul(ll x, ll y, ll m){
        ll res = x * y - (ll)((ld)x * y / m + 0.5) * m;
        return res < 0 ? res + m : res;
    }
    inline ll mod_pow(ll x, ll n, ll m){
        ll res = 1 % m;
        for (; n; n >>= 1){
            if (n & 1) res = mod_mul(res, x, m);
            x = mod_mul(x, x, m);
        }
        return res;
    }
}

// O(k logn logn logn), k = number of rounds
    performed
inline bool miller_rabin(ll n){
    if (n <= 2 || (n & 1 ^ 1)) return (n == 2);

```

```

if (n < MAXP) return spf[n] == n;
ll c, d, s = 0, r = n - 1;
for (; !(r & 1); r >= 1, s++) {}
// each iteration is a round
for (int i = 0; primes[i] < n && primes[i] < 32; i++){
    c = mod_pow(primes[i], r, n);
    for (int j = 0; j < s; j++){
        d = mod_mul(c, c, n);
        if (d == 1 && c != 1 && c != (n - 1)) return false;
    }
    c = d;
    if (c != 1) return false;
}
return true;
}

inline void init(){
    int i, j, k, cnt = 0;
    for (i = 2; i < MAXP; i++){
        if (!spf[i]) primes[cnt++] = spf[i] = i;
        for (j = 0, k; (k = i * primes[j]) < MAXP; j++){
            spf[k] = primes[j];
            if (spf[i] == spf[k]) break;
        }
    }
}

// Expected complexity  $O(n^{1/4})$ 
ll pollard_rho(ll n){
    while (1){
        ll x = rng() % n, y = x, c = rng() % n, u = 1, v,
            t = 0;
        ll *px = seq, *py = seq;
        while (1){
            *py++ = y = mod_add(mod_mul(y, y, n), c, n);
            *py++ = y = mod_add(mod_mul(y, y, n), c, n);
            if ((x = *px++) == y) break;
            v = u;
            u = mod_mul(u, abs(y - x), n);
            if (!u) return __gcd(v, n);
            if (++t == 32){
                t = 0;
                if ((u = __gcd(u, n)) > 1 && u < n) return u;
            }
        }
        if (t && (u = __gcd(u, n)) > 1 && u < n) return u;
    }
}

vector<ll> primeFactorize(ll n){
    if (n == 1) return vector<ll>();
    if (miller_rabin(n)) return vector<ll>{n};
    vector<ll> v, w, vw;
    while (n > 1 && n < MAXP){

```

```

        vw.push_back(spf[n]);
        n /= spf[n];
    }
    if (n >= MAXP) {
        ll x = pollard_rho(n);
        v = primeFactorize(x);
        w = primeFactorize(n / x);
        vw.resize( (int)v.size() + (int)w.size() );
        merge( v.begin(), v.end(), w.begin(), w.end(),
            vw.begin() );
    }
    return vw;
}
}

```

1.17.7 sieve

```

const int MAXN = 1e6+5;
vector<int> prime;
bool is_composite[MAXN];
void sieve () {
    for (int i = 2; i < MAXN; ++i) {
        if (!is_composite[i]) prime.push_back(i);
        for (int j = 0; j < prime.size() && i * prime[j] <
            MAXN; ++j) {
            is_composite[i * prime[j]] = true;
            if (i % prime[j] == 0) break;
        }
    }
}

```

1.18 strings

1.18.1 aho corasick

```

//Aho Corasick -  $O(N)$ ;  $N$  = sum of lengths of all
// strings in the set;

// Vertex means state, state means vertex. They are
// used interchangeably;
/**
1. Proper suffix of root vertex is the root itself. So
   its suffix link will point to itself;
2. Suffix link of vertex of depth 1 will point to the
   root;
3. root vertex is 0;
**/

```

```

const int ALPHA = 26; //max no. of alphabets
struct AhoCorasick{
    struct Vertex {
        int edge[ALPHA]; // edges indicate letters of
            alphabet;
        bool leaf = false; // leaf = true- means this
            state represents a string of the set;
        int p = -1; // parent of this state;
        char pch; // this state's parent ----pch----> this
            state;
        int link = -1; // suffix link of this state; //
            link = -1 means suffix link has not been
            determined;
        int go[ALPHA]; // go to next state through a char
            edge;
        Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
            memset( edge, -1, sizeof(edge) );
            memset( go, -1, sizeof(go) );
        }
    };

    vector<Vertex> trie;
    AhoCorasick() : trie(1) {}

    void add_string(string const& s) {
        int node = 0;
        for (char ch : s) {
            int ch_idx = ch - 'a';
            if (trie[node].edge[ch_idx] == -1) {
                trie[node].edge[ch_idx] = trie.size();
                trie.push_back( Vertex(node, ch) );
            }
            node = trie[node].edge[ch_idx];
        }
        trie[node].leaf = true;
    }

    int get_link(int node) { // get suffix link;
        if (trie[node].link == -1) {
            if (node == 0 || trie[node].p == 0)
                trie[node].link = 0;
            else
                trie[node].link = go(get_link(trie[node].p),
                    trie[node].pch);
        }
        return trie[node].link;
    }

    int go(int node, char ch) { // go to next state;
        int ch_idx = ch - 'a';
        if (trie[node].go[ch_idx] == -1) {
            if (trie[node].edge[ch_idx] != -1)
                trie[node].go[ch_idx] = trie[node].edge[ch_idx];
        }
    }
}

```

```

else
    trie[node].go[ch_idx] = (node == 0) ? 0 :
        go(get_link(node), ch);
}
return trie[node].go[ch_idx];
}
};

```

1.18.2 manachers

```

void manachers(string s, vector<int> &d1, vector<int>
&d2) {
    int n = s.size();
    /// odd length palindromes.
    d1.resize(n);
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
        while (0 <= i - k && i + k < n && s[i - k] == s[i +
            k]) k++;
        d1[i] = k--;
        if (i + k > r) {
            l = i - k;
            r = i + k;
        }
    }
    /// even length palindromes. suppose "abba" is a
    palindrome. Here, 2nd index(0-based indexing) is
    the center.
    d2.resize(n);
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i
            + 1);
        while (0 <= i - k - 1 && i + k < n && s[i - k - 1]
            == s[i + k]) k++;
        d2[i] = k--;
        if (i + k > r) {
            l = i - k - 1;
            r = i + k;
        }
    }
}

```

1.18.3 suffix array(better version by Ashiquul)

```

/**
1. sa[i] = i'th suffix, i from 0 to n-1
2. everything is in 0'th base

```

```

3. lcp[i] = lcp of (i-1)th and ith suffix, i from 0 to
n-1. (In the code, height[] is the lcp array)
4. adjust the alpha, usually for string ALPHA = 128
(max ascii value)
5. notice if clearing may cause tle
6. remove range_lcp_init() if not required
7. rev_sa[sa[i]] = i;
**/
const int MAXN = 1010; /// always take 10 extra.
const int ALPHA = 256, LOG = 12; ///LOG is log2(MAXN)+3
struct SuffixArray {
    int sa[MAXN], data[MAXN], rnk[MAXN], height[MAXN], n;
    int wa[MAXN], wb[MAXN], wws[MAXN], wv[MAXN];
    int lg[MAXN], rmq[MAXN][LOG], rev_sa[MAXN];
    int cmp(int *r, int a, int b, int l) {
        return (r[a] == r[b]) && (r[a + l] == r[b + l]);
    }
    void DA(int *r, int *sa, int n, int m) {
        int i, j, p, *x = wa, *y = wb, *t;
        for (i = 0; i < m; i++) wws[i] = 0;
        for (i = 0; i < n; i++) wws[x[i] = r[i]]++;
        for (i = 1; i < m; i++) wws[i] += wws[i - 1];
        for (i = n - 1; i >= 0; i--) sa[--wws[x[i]]] = i;
        for (j = 1, p = 1; p < n; j *= 2, m = p) {
            for (p = 0, i = n - j; i < n; i++) y[p++] = i;
            for (i = 0; i < n; i++) if (sa[i] >= j) y[p++] = sa[i] - j;
            for (i = 0; i < n; i++) wv[i] = x[y[i]];
            for (i = 0; i < n; i++) wws[i] = 0;
            for (i = 0; i < n; i++) wws[wv[i]]++;
            for (i = 1; i < m; i++) wws[i] += wws[i - 1];
            for (i = n - 1; i >= 0; i--) sa[--wws[wv[i]]] = y[i];
            for (t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i < n; i++)
                x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;
        }
    }
    void calheight(int *r, int *sa, int n) {
        int i, j, k = 0;
        for (i = 1; i < n; i++) rnk[sa[i]] = i;
        for (i = 0; i < n; height[rnk[i + 1]] = k)
            for (k ? k-- : 0, j = sa[rnk[i] - 1]; r[i + k] == r[j + k]; k++);
    }
    void suffix_array (string &A) {
        n = A.size();
        for (int i = 0; i < max(n + 5, ALPHA); i++)
            sa[i] = data[i] = rnk[i] = height[i] = wa[i] = wb[i] = wws[i] = wv[i] = 0;
        for (int i = 0; i < n; i++) data[i] = A[i];
        DA(data, sa, n + 1, ALPHA);
        calheight(data, sa, n);
        for (int i = 0; i < n; i++) sa[i] = sa[i + 1],
            height[i] = height[i + 1], rev_sa[sa[i]] = i;
        range_lcp_init();
    }
}
/** LCP for range : build of rmq table **/

```

```

void range_lcp_init() {
    for (int i = 0; i < n; i++) rmq[i][0] = height[i];
    for (int j = 1; j < LOG; j++) {
        for (int i = 0; i < n; i++) {
            if (i + (1 << j) - 1 < n) rmq[i][j] =
                min(rmq[i][j - 1], rmq[i + (1 << (j - 1))][j - 1]);
            else break;
        }
    }
    lg[0] = lg[1] = 0;
    for (int i = 2; i <= n; i++) {
        lg[i] = lg[i / 2] + 1;
    }
}
/** lcp between l'th to r'th suffix **/
int query_lcp(int l, int r) {
    assert(l <= r);
    assert(l >= 0 && l < n && r >= 0 && r < n);
    if (l == r) return n - sa[l];
    l++;
    int k = lg[r - l + 1];
    return min(rmq[l][k], rmq[r - (1 << k) + 1][k]);
}
}SA;

/// substring sort comparator function. (it is used to
sort all possible substrings of a string)
bool cmp(pair<int, int> &lhs, pair<int, int> &rhs) {
    int l1 = lhs.first, r1 = lhs.second, l2 = rhs.first,
        r2 = rhs.second;
    bool f = 0;
    if (SA.rev_sa[l2] < SA.rev_sa[l1]) {
        swap(l1, l2);
        swap(r1, r2);
        f ^= 1;
    }
    int len1 = r1 - l1 + 1, len2 = r2 - l2 + 1;
    int com = SA.query_lcp(SA.rev_sa[l1], SA.rev_sa[l2]);
    if (com < min(len1, len2)) return f ^ 1;
    return (len1 < len2) ^ f;
}

int main() {
    string s;
    cin >> s;
    SA.suffix_array(s);
    SA.range_lcp_init();
    vector<pair<int, int>> sub_string;
    for (int i = 0; i < s.size(); ++i) {
        for (int j = i; j < s.size(); ++j) {
            sub_string.PB(MP(i, j));
        }
    }
}

```

```

    sort(sub_string.begin(), sub_string.end(), cmp);
    ///substring sorted.
}

```

1.18.4 z-algorithm

```

//O(N)
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for( int i = 1, l = 0, r = 0; i < n; ++i ) {
        if (i <= r) z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    return z;
}

```

1.19 ternary search

// complexity: $O(\lg N)$
 // Be careful about choosing the boundary

/**
 What is Unimodal function?
 -We are given a function $f(x)$ which is unimodal on an interval $[l, r]$. By unimodal function, we mean one of two behaviors of the function:
 1. The function strictly increases first, reaches a maximum (at a single point or over an interval), and then strictly decreases.
 2. The function strictly decreases first, reaches a minimum, and then strictly increases.
 **/

/// Ternary search on function f

/**
 This part is an unproven concept. Invented by me. Use it at your own risk.
 If the extrema of the unimodal function is a segment instead of a point then to get the leftmost or rightmost extremum return value of the function we may consider adding \geq instead of $>$ in the "if()" part of the code only if necessary.
 **/

```

//1. Finding the minimum function (doubles precision)
int steps = 200;
void ternary_search() {
    double lo = -inf, hi = inf, mid1, mid2;
    for(int i = 0; i < steps; i++) {
        mid1 = (lo*2+hi) / 3.0;
        mid2 = (lo+2*hi) / 3.0;
        if(f(mid1) > f(mid2)) lo = mid1;
        else hi = mid2;
    }
    double ans = f(lo);
    return ans;
}

```

```

//2. Finding the maximum function (doubles precision)
void ternary_search() {
    double lo = -inf, hi = inf, mid1, mid2;
    for(int i = 0; i < steps; i++) {
        mid1 = (lo*2+hi) / 3.0;
        mid2 = (lo+2*hi) / 3.0;
        if(f(mid1) > f(mid2)) hi = mid2;
        else lo = mid1;
    }
    double ans = f(lo);
    return ans;
}

```

```

//3. Finding the minimum function (search on integer range)
void ternary_search() {
    int lo = -inf, hi = inf, mid1, mid2;
    while(hi - lo > 4) {
        mid1 = (lo + hi) / 2;
        mid2 = (lo + hi) / 2 + 1;
        if(f(mid1) > f(mid2)) lo = mid1;
        else hi = mid2;
    }
    ans = inf;
    for(int i = lo; i <= hi; i++) ans = min(ans, f(i));
}

//4. Finding the maximum function (search on integer range)
void ternary_search() {
    int lo = -inf, hi = inf, mid1, mid2;
    while(hi - lo > 4) {
        mid1 = (lo + hi) / 2;
        mid2 = (lo + hi) / 2 + 1;
        if(f(mid1) > f(mid2)) hi = mid2;
        else lo = mid1;
    }
    ans = inf;
    for(int i = lo; i <= hi; i++) ans = max(ans, f(i));
}

```

2 omar bhai

2.1 Data Structure

2.1.1 2D Fenwick tree

```

int sum[10][10], row,col;
void update(int x,int y,int val){
    while(x<=row){
        while(y<=col){
            sum[x][y]+=val;
            y+=y&(-y);
        }
        x+=x&(-x);
    }
}

int getSum(int x,int y){
    int res=0;
    while(x){
        while(y){
            res+=sum[x][y];
            y-=y&(-y);
        }
        x-=x&(-x);
    }
    return res;
}

```

2.2 Hashing

```

//***Double Hashing***
//***Complexity:  $O(N)$ 
struct simpleHash{
    vector<long long>p;
    vector<long long>h;
    long long base, len, mod = (long long)1e9+7;
    simpleHash(){}
    simpleHash(string &str, long long b){
        base = b; len = str.size(); //0 base index str
        p.resize(len,1);
        h.resize(len+1,0);
        for(int i=1; i<len; i++) p[i] = ( p[i-1]*base )%mod;
        for(int i=1; i<=len; i++) h[i] = (h[i-1]*base +
            (str[i-1]-'a' + 3) )%mod;
    }
    long long rangeHash(int l,int r){ //l and r inclusive
        and also 0 based
        return ( h[r+1]- ( h[l] * p[r-l+1] ) % mod ) + mod ) %
            mod;
    }
}

```

```
};
struct doubleHashing{
    simpleHash h1,h2;
    doubleHashing(string &str){
        h1 = simpleHash(str, 149);
        h2 = simpleHash(str, 223);
    }
    long long rangeHash(int l,int r){ //l and r inclusive
        and also 0 based
        return ( h1.rangeHash(l,r)<<32LL) ^
            h2.rangeHash(l,r);
    }
};
```

2.3 Number theory

2.3.1 0Number theory rules

1. If $N = (p_1^{a_1}) * (p_2^{a_2}) * \dots * (p_N^{a_N})$ then

$$\text{NOD}(N) = (a_1 + 1) * (a_2 + 1) * \dots * (a_N + 1)$$
2. If $N = (p_1^{a_1}) * (p_2^{a_2}) * \dots * (p_N^{a_N})$ then,

$$\text{SOD}(N) = ((p_1^{a_1+1}) - 1) / (p_1 - 1) * \dots * ((p_N^{a_N+1}) - 1) / (p_N - 1)$$
1. If M & N are co-prime then the formula holds ::

$$\text{Phi}(M) * \text{Phi}(N) = \text{Phi}(M * N)$$
2. The Numbers(a) less than or Equal to N who all have $[\text{gcd}(a, N) = d]$ will be $\text{Phi}(N/d)$
3. (sum of all the $[\text{Phi}(d)] = N$) ; where d represents all the divisors of N
4. For $N > 2$ $\text{Phi}(N)$ is always Even
5. Sum of all the Numbers less than or Equal to N that are Co Prime with N is $[N * \text{Phi}(N) / 2]$
6. $[\text{Lcm}(1, n) + \text{Lcm}(2, n) + \dots + \text{Lcm}(1, n)] = (n / 2) * (\text{sum of all } \text{Phi}(d) * 1)$;
 d is the divisors of n
1. Mod Inverse can be solved recursively with the following formula ::

$$\text{inv}[a] = (-\text{Floor}(\text{Mod} / A) * \text{inv}[\text{Mod} \% A] + \text{Mod}) \% \text{Mod}$$
1. $\log_B(x) = (\log_C(x) / \log_C(B))$, [Here , $\log_B(x)$ means $\log(x)$ based B] ;
2. What does $\log_{10}(X)$ means? $10^{(\text{fractional part of the result})}$ means the leading digit!!!
3. $\lgamma(x) = \log(1) + \dots + \log(x-1)$ // this is like magic!!

2.3.2 Chinese remainder theorem

```

//*** Finding the x that satisfies all the congruence
equation***
//*** Complexity:  $O(N \log(N))$ ***
/*
basically chinese remainder theorem is solved using
extended euclid algorithm.
the problems can be converted to a diophantine
equation of  $ax+by=c$  format.
 $x=a1(\text{mod } n1)$ 
 $x=a2(\text{mod } n2)$ 
we can rewrite these equations as follow:
 $x=a1+n1*k1$ 
 $x=a2+n2*k2$ 
from the above we can write
 $a1+n1*k1 = a2+n2*k2$ 
or,  $n1*(-k1) + n2*k2 = a1-a2$  [which is a diophantine
equation. so the solution will exists if and
only if  $(a1-a2)\%gcd(n1,n2)==0$ ]
[changing the equation we get ::  $x = a1 +$ 
 $x'*[(a2-a1)/d]*n1$  where  $d$  is the gcd of  $n1$  and
 $n2$ ,  $x'$  is the value we get from extended euclid]
*/
struct ChineseRemainderTheorem{
    long long ans,lcm;
    bool haveSolution;
    ChineseRemainderTheorem(){}
    ChineseRemainderTheorem(vector<long
        long>&a,vector<long long>&n){
        haveSolution=true;
        ans=a[0];
        lcm=n[0];
        for(int i=1;i<n.size();i++){
            long long x,y;
            long long gcd=extendedEuclid(lcm,n[i],x,y);
            if((a[i]-ans)%gcd){
                haveSolution=false;
                return;
            }
            ans=(ans + ( ( x * ((a[i]-ans) /gcd) )
                %n[i]/gcd) ) * lcm);
            lcm=LCM(lcm,n[i],gcd);
            ans=((ans%lcm)+lcm)%lcm;
        }
        //so ans variable holds the required result that
        will satisfy all the congruence equations.
    }
    long long extendedEuclid(long long a,long long b,long
        long &x,long long &y){
        if(!b){
            x=1; y=0;

```

```

        return a;
    }
    long long x0,y0;
    long long gcd=extendedEuclid(b,a%b,x0,y0);
    x=y0;
    y=x0-(a/b)*y0;
    return gcd;
}

long long LCM(long long x,long long y,long long gcd){
    return (x/gcd)*y;
}

};
```

2.3.3 Linear Diophantine equation

```

//*** Linear Diophantine[ax+by=c] one actual solution
and solution count between range (x1,x2) &
(y1,y2)***
//*** Complexity: O(log N)***
/*
1. Linear Diophantine Equation [ ax+by=c ] will have
solution only if [ c % gcd(a,b) == 0];
2. The other solutions of [ax+by=c] are for any integer
value of k [ x + k*(b/gcd(a,b)) , y -
k*(a/gcd(a,b))]
3. The largest number that can't be expressed in ax+by
form where ( [a,b,x,y all are >=0] and a,b are
co-prime) is [ab-a-b].
4. All multiples of gcd(a, b) greater than [lcm(a,
n)-a-n] are representable in the form [ax+by]
where [a,b,x,y all are >=0]
5. There are exactly [(a-1)*(b-1)/2] values that can't
be expressed in ax+by where [a,b,x,y all are >=0]
6. Compare [ax + by = gcd(a,b)] with [ ( a/b )x + by =
gcd( a,b )] or [ ( a - (Floor( a/b ) * b) )x + by =
gcd( a,b )] For Recursion.
7. Extended Euclid Algorithm finds the x, y and gcd(
a,b ) of Bzouts lemma [ ax + by = gcd( a,b ) ]
*/
//***important :: this algo will not work if [a==0 ||
b==0], we will have to compute the result
separately, in those case***
struct LinearDiophantineEquation{
int extendedEuclid(int a,int b,long long &x,long long
&y){
if(!b){
x=1,y=0;
return a;
}
long long x0,y0;
int gcd=extendedEuclid(b,a%b,x0,y0);

```



```

    x=y0;
    y=x0-(a/b)*y0;
    return gcd;
}
bool getOneSolution(int a,int b,int c,long long
    &x,long long &y,long long &gcd){
    gcd=extendedEuclid(abs(a),abs(b),x,y);
    if(c%gcd)return false;
    x*=(c/gcd);
    y*=(c/gcd);
    if(a<0)x*=-1;
    if(b<0)y*=-1;
    return true;
}
void shiftResult(int a,int b,long long &x,long long
    &y,long long cnt){
    x+=b*cnt; //this can cause overflow in int
    y-=a*cnt; //this can cause overflow in int
}
int getAllSolution(int a,int b,int c,int xLow,int
    xHigh,int yLow,int yHigh){
    long long x,y,gcd;
    if(!getOneSolution(a,b,c,x,y,gcd))return 0;
    a/=gcd;
    b/=gcd;
    int sign_a=(a>0)?+1:-1;
    int sign_b=(b>0)?+1:-1;
    shiftResult(a,b,x,y,(xLow-x)/b);
    if(x<xLow)shiftResult(a,b,x,y,sign_b);
    if(x>xHigh)return 0;
    long long lx1=x;
    shiftResult(a,b,x,y,(xHigh-x)/b);
    long long rx1=x;
    shiftResult(a,b,x,y,-(yLow-y)/a);
    if(y<yLow)shiftResult(a,b,x,y,-sign_a);
    if(y>yHigh)return 0;
    long long lx2=x;
    shiftResult(a,b,x,y,-(yHigh-y)/a);
    long long rx2=x;
    if(lx2>rx2)swap(lx2,rx2);
    long long lx=max(lx1,lx2);
    long long rx=min(rx1,rx2);
    if(lx>rx)return 0;
    return (rx-lx)/abs(b) +1;
}
};

```

2.3.4 Sum of NOD of numbers from 1 to N

```

//*** Complexity: O( sqrt(N) )***
long long SNOD(long long n=10){
    long long res=0;
    long long len=sqrtl(n);
    for(int i=1;i<=len;i++)res+=(n/i)-i; //finding the
        ordered pair like a<b and a*b<n
    res*=2LL; //converting pair count to single value
    res+=len; //adding the equal values like (1,1),
        (2,2)..
    return res;
}

```

2.4 Rotating Calipers

```

//***Complexity: O(N)
struct RotatingCalipers{
    long double res=1e18;
    int len;
    /*this is just a variation of rotating calipers
        finding the minimum distance of two parallel
        line in which the polygon can pass through*/
    RotatingCalipers(vector<pair<int,int> >&hull){
        //convex hull.
        len=hull.size();
        for(int i=0,j=i+1;i<len;i++){
            pair<int,int>&p1=hull[i];
            pair<int,int>&p2=hull[(i+1)%len];
            while(area(p1, p2, hull[j%len]) < area(p1, p2,
                hull[(j+1)%len]))j+=1;
            res=min(res, dist(p1, p2, hull[j%len]));
        }
    }
    //Cartesian Distance between point p1 & p2
    typedef pair<long double, long double> pld;
    long double dist(pld p1,pld p2){
        long double xDist=p1.first-p2.first;
        long double yDist=p1.second-p2.second;

```

```

        return sqrtl((xDist*xDist)+(yDist*yDist));
    }
    typedef pair<long long, long long> pll;
    //returns area*2.0 enclosed by p1, p2, p3 points
    long long area(pll p1, pll p2, pll p3){
        return llabs(p1.first*p2.second +
            p2.first*p3.second + p3.first*p1.second
            -p1.second*p2.first-p2.second*p3.first-p3.second*p1.first);
    }
    //distance from p3 to line going through p1 & p2
    long double dist(pld p1, pld p2, pld p3){
        long double up=(p2.second-p1.second)*p3.first
            -(p2.first-p1.first)*p3.second
            +p2.first*p1.second
            -p1.first*p2.second;
        long double down=sqrtl(
            (p2.second-p1.second)*(p2.second-p1.second)
            +(p2.first-p1.first)*(p2.first-p1.first));
        return fabs1(up/down);
    }
};

```

3 shaad bhai

3.1 Intersection of Two Path

```

//Calculating common path between node (a,b) and (c,d)
int main() {
    scanf("%d %d %d %d",&a,&b,&c,&d);
    int t1=lca(a,b),t2=lca(c,d);
    if (dep[t1]>dep[t2]) {
        swap(a,c); swap(b,d); swap(t1,t2);
    }
    int p1=lca(a,c),p2=lca(b,d);
    int p3=lca(a,d),p4=lca(b,c);
    int k1,k2;
    if (p1==t1) k1=p4; else k1=p1;
    if (p2==t1) k2=p3; else k2=p2;
    if (dis(k1,k2)==1&&t2!=k1) // no intersection
        else // the path is (k1,k2)
}

```