ACM ICPC NOTEBOOK
Team : No_Handle
Content

Ford Fulkerson

```
//   Ford Fulkerson's Maximum Flow
//  Time Complexity : O(E * FLOW)
vector<int> v[MAX];
int cap[MAX][MAX], S, D, par[MAX];
bool vis[MAX];
bool bfs()
{
   par[S] = -1;
   queue<int> q;
   q.push(S);
   MS0(vis);
   vis[S] = 1;
   while (!q.empty())
   {
              int from = q.front();
              q.pop();
              for (int i = 0; i < v[from].size(); i++)
              {
                          int to = v[from][i];
                          if (vis[to] || cap[from][to] <= 0)
                                      continue;
                          q.push(to);
                          vis[to] = 1;
                          par[to] = from;
              }
   }
   return vis[D];
}
int ford_fulkerson()
{
   int F = 0;
   while (bfs())
   {
              int flow = INF;
              for (int v = D, u = par[D]; v != S; v = par[v], u = par[u])
                          flow = min(flow, cap[u][v]);

              for (int v = D, u = par[D]; v != S; v = par[v], u = par[u])
              {
                          cap[u][v] -= flow;
                          cap[v][u] += flow;
              }
              F += flow;
   }
   return F;
}
```

Dinic's Flow

```
// Dinic's MAX FLOW
// Time Complexity : O(V^2E)
int n, S = 0, D, cap[MAX][MAX], dis[MAX], work[MAX];
int M, V;
vector<int> v[MAX];

bool bfs()    //    To check if an augmented path exists
{             //    Thus finding shortest augmented path
   for (int i = 0; i <= D; i++)
             dis[i] = INF;
   dis[S] = 0;
   queue<int> q;
   q.push(S);
   while (!q.empty())
   {
             int cur = q.front();
             q.pop();
             for (int i : v[cur])
             {
                        if (dis[i] != INF || cap[cur][i] <= 0)
                                   continue;
                        dis[i] = dis[cur] + 1;
                        q.push(i);
             }
   }
   return (dis[D] != INF);
}

int dfs(int cur, int flow)    //    traversing augmented path and
{                             //    updating the flow
   if (cur == D)
             return flow;
   for (int &i = work[cur]; i < v[cur].size(); i++)
   {
             int to = v[cur][i];
             if (dis[to] != dis[cur] + 1 || cap[cur][to] <= 0)
                        continue;
             int f = dfs(to, min(flow, cap[cur][to]));
             if (f > 0)    //    augmented path found
             {
                        cap[cur][to] -= f;
                        cap[to][cur] += f;
                        return f;
             }
   }
   return 0;
}

int max_flow()
{
   //    S : Source, D : Destination
   int F = 0;    //    Total Flow
   while (bfs())
   {
             MS0(work);
             while (1)    //    covering all the augmented paths
             {            //    marked by the bfs
                        int cur = dfs(S, INF);
                        if (cur == 0)    //    all augmented paths covered
                                   break;
                        F += cur;
             }
   }
   return F;
}
```

Push Relabel Max-Flow

```
// Running time:    O(|V|^3)
typedef long long LL;
struct Edge {
   int from, to, cap, flow, index;
   Edge(int from, int to, int cap, int flow, int index) :
             from(from), to(to), cap(cap), flow(flow), index(index) {}
};
struct PushRelabel {
   int N;
   vector<vector<Edge> > G;
   vector<LL> excess;
   vector<int> dist, active, count;
   queue<int> Q;
   PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2 * N) {}
   void AddEdge(int from, int to, int cap) {
             G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
             if (from == to) G[from].back().index++;
             G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
   }
   void Enqueue(int v) {
             if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }
   }
   void Push(Edge &e) {
             int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
             if (dist[e.from] <= dist[e.to] || amt == 0) return;
             e.flow += amt;
             G[e.to][e.index].flow -= amt;
             excess[e.to] += amt;
             excess[e.from] -= amt;
             Enqueue(e.to);
   }
```

```
void Gap(int k) {
    for (int v = 0; v < N; v++) {
        if (dist[v] < k) continue;
        count[dist[v]]--;
        dist[v] = max(dist[v], N + 1);
        count[dist[v]]++;
        Enqueue(v);
    }
}
void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2 * N;
    for (int i = 0; i < G[v].size(); i++)
        if (G[v][i].cap - G[v][i].flow > 0)
            dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    Enqueue(v);
}
void Discharge(int v) {
    for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
    if (excess[v] > 0) {
        if (count[dist[v]] == 1)
            Gap(dist[v]);
        else
            Relabel(v);
    }
}
LL GetMaxFlow(int s, int t) {
    count[0] = N - 1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
        excess[s] += G[s][i].cap;
        Push(G[s][i]);
    }
    while (!Q.empty()) {
        int v = Q.front();
        Q.pop();
        active[v] = false;
        Discharge(v);
    }
    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
}
};
```

Hopcroft-Karp Maximum Matching

```
//    Hopcroft-Karp Maximum Matching
//  Time Complexity : O(sqrt(V) * E)
#define NIL 0
int n, matchL[MAX], matchR[MAX], dis[MAX];
vector<int> v[MAX];
//    Vertices are numbered from 1 to n

bool bfs()    //    Checking if an augmented path exists
{                         //    or not and marking it
   queue<int> q;
   for (int i = 1; i <= n; i++)
   {
        if (matchL[i] == NIL)
        {
            dis[i] = 0;
            q.push(i);
        }
        else
            dis[i] = INF;
   }
   dis[NIL] = INF;
   while (!q.empty())
   {
        int from = q.front();
        q.pop();
        if (from == NIL)
            continue;
        for (int i = 0; i < v[from].size(); i++)
        {
            int to = v[from][i];
            if (dis[matchR[to]] == INF)
            {
                dis[matchR[to]] = dis[from] + 1;
                q.push(matchR[to]);
            }
        }
   }
   return (dis[NIL] != INF);
}

bool dfs(int from)    //    Finding the marked path and
{                                          //    updating the matching
   if (from == NIL)
        return 1;
   for (int i = 0; i < v[from].size(); i++)
   {
        int to = v[from][i];
        if (dis[matchR[to]] == dis[from] + 1)
        {
            if (dfs(matchR[to]))
            {
                matchR[to] = from;
```

```
                    matchL[from] = to;
                    return 1;
                }
            }
   }
   return 0;
}

int max_matching()
{
   MS0(matchR);    //    Matching array for set at right side
   MS0(matchL);    //    Matching array for set at left side
   int matching = 0;    //    Total matching
   while (bfs())
   {
        for (int i = 1; i <= n; i++)    //    For each vertex finding a match
        {
            if (matchL[i] == NIL && dfs(i))
                matching++;
        }
   }
   return matching;
}
```

Min Cost Max Flow

```
/*
        Note that MCMF routine is taken from http://shygypsy.com/tools/mcmf4.cpp.
*/
/* ALL COSTS MUST BE NON-NEGATIVE!
 * COMPLEXITY:  Worst case: O(m*log(m)*flow  <?  n*m*log(m)*fcost)
 **/
// adjacency matrix (fill this up)
int cap[MAX][MAX];

// cost per unit of flow matrix (fill this up)
int cost[MAX][MAX];

// flow network and adjacency list
int fnet[MAX][MAX], adj[MAX][MAX], deg[MAX];

// Dijkstra's predecessor, depth and priority queue
int par[MAX], d[MAX], q[MAX], inq[MAX], qs;

// Labelling function
int pi[MAX];
#define BUBL { \
        t = q[i]; q[i] = q[j]; q[j] = t; \
        t = inq[q[i]]; inq[q[i]] = inq[q[j]]; inq[q[j]] = t; }

// Dijkstra's using non-negative edge weights (cost + potential)
#define Pot(u,v) (d[u] + pi[u] - pi[v])
bool dijkstra( int n, int s, int t )
{
        CLR( d, 0x3F );
        CLR( par, -1 );
        CLR( inq, -1 );
        //for( int i = 0; i < n; i++ ) d[i] = Inf, par[i] = -1;
        d[s] = qs = 0;
        inq[q[qs++] = s] = 0;
        par[s] = n;

        while( qs )
        {
        // get the minimum from q and bubble down
        int u = q[0]; inq[u] = -1;
        q[0] = q[--qs];
        if( qs ) inq[q[0]] = 0;
        for( int i = 0, j = 2*i + 1, t; j < qs; i = j, j = 2*i + 1 )
        {
        if( j + 1 < qs && d[q[j + 1]] < d[q[j]] ) j++;
        if( d[q[j]] >= d[q[i]] ) break;
        BUBL;
        }

        // relax edge (u,i) or (i,u) for all i;
        for( int k = 0, v = adj[u][k]; k < deg[u]; v = adj[u][++k] )
        {
        // try undoing edge v->u
        if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u] )
        d[v] = Pot(u,v) - cost[v][par[v] = u];

        // try using edge u->v
        if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v] )
        d[v] = Pot(u,v) + cost[par[v] = u][v];

        if( par[v] == u )
        {
        // bubble up or decrease key
        if( inq[v] < 0 ) { inq[q[qs] = v] = qs; qs++; }
        for( int i = inq[v], j = ( i - 1 )/2, t;
        d[q[i]] < d[q[j]]; i = j, j = ( i - 1 )/2 )
        BUBL;
        }
        }
        }

        for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i] += d[i];

        return par[t] >= 0;
}
#undef Pot
```

```
int mcmf4( int n, int s, int t, int &fcost )
{
	// build the adjacency list
	CLR( deg, 0 );
	for( int i = 0; i < n; i++ )
	for( int j = 0; j < n; j++ )
	if( cap[i][j] || cap[j][i] ) adj[i][deg[i]++] = j;

	CLR( fnet, 0 );
	CLR( pi, 0 );
	int flow = fcost = 0;

	// repeatedly, find a cheapest path from s to t
	while( dijkstra( n, s, t ) )
	{
	// get the bottleneck capacity
	int bot = INT_MAX;
	for( int v = t, u = par[v]; v != s; u = par[v = u] )
	bot <?= fnet[v][u] ? fnet[v][u] : ( cap[u][v] - fnet[u][v] );

	// update the flow network
	for( int v = t, u = par[v]; v != s; u = par[v = u] )
	if( fnet[v][u] ) { fnet[v][u] -= bot; fcost -= bot * cost[v][u]; }
	else { fnet[u][v] += bot; fcost += bot * cost[u][v]; }

	flow += bot;
	}
	return flow;
}

// For dense graph use the following djikstra
bool dijkstra( int n, int s, int t )
{
	for( int i = 0; i < n; i++ ) d[i] = Inf, par[i] = -1;
	d[s] = 0;
	par[s] = -n - 1;

	while( 1 )
	{
	// find u with smallest d[u]
	int u = -1, bestD = Inf;
	for( int i = 0; i < n; i++ ) if( par[i] < 0 && d[i] < bestD )
	bestD = d[u = i];
	if( bestD == Inf ) break;

	// relax edge (u,i) or (i,u) for all i;
	par[u] = -par[u] - 1;
	for( int i = 0; i < deg[u]; i++ )
	{
	// try undoing edge v->u
	int v = adj[u][i];
	if( par[v] >= 0 ) continue;
	if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u] )
	d[v] = Pot( u, v ) - cost[v][u], par[v] = -u-1;

	// try edge u->v
	if( fnet[u][v] < cap[u][v] && d[v] > Pot(u,v) + cost[u][v] )
	d[v] = Pot(u,v) + cost[u][v], par[v] = -u - 1;
	}
	}

	for( int i = 0; i < n; i++ ) if( pi[i] < Inf ) pi[i] += d[i];

	return par[t] >= 0;
}
```

Extended Euclid’s GCD

```
//   Extended Euclid's GCD
struct node
{
   ll a, b, c;
   node(ll x = 0, ll y = 0, ll z = 0)
   {
		a = x; // gcd
		b = y; // x
		c = z; // y
   }
};

node xgcd(ll a, ll b) // extended euclid gcd  ax + by = g
{
   if (b == 0)
		return node(a, 1, 0);

   node tmp = xgcd(b, a % b);
   node ans(tmp.a, tmp.c, tmp.b - ((a / b) * tmp.c));
   return ans;
}
```

Fast Fourier Transformation

```
const double PI = 4 * atan(1);
typedef long long LL;
const int MAX = 2e5 + 5;
const int prime = 13313;
namespace FFT {
// super optimised fft code
const int N = 20;
const int MAXN = (1 << N);
class cmplx {
private:
```

```
		double x, y;
public:
		cmplx () : x(0.0), y(0.0) {}
		cmplx (double a) : x(a), y(0.0) {}
		cmplx (double a, double b) : x(a), y(b) {}
		double get_real() { return this->x; }
		double get_img() { return this->y; }
		cmplx conj() { return cmplx(this->x, -(this->y)); }
		cmplx operator = (const cmplx& a) { this->x = a.x; this->y = a.y; return *this; }
		cmplx operator + (const cmplx& b) { return cmplx(this->x + b.x, this->y + b.y); }
		cmplx operator - (const cmplx& b) { return cmplx(this->x - b.x, this->y - b.y); }
		cmplx operator * (const double& num) { return cmplx(this->x * num, this->y * num); }
		cmplx operator / (const double& num) { return cmplx(this->x / num, this->y / num); }
		cmplx operator * (const cmplx& b) {
		return cmplx(this->x * b.x - this->y * b.y, this->y * b.x + this->x * b.y);
		}
		cmplx operator / (const cmplx& b) {
		cmplx temp(b.x, -b.y); cmplx n = (*this) * temp;
		return n / (b.x * b.x + b.y * b.y);
		}
};
cmplx w[MAXN];
cmplx f1[MAXN];
int rev[MAXN];
void ReserveBits(int k) {
		static int rk = -1, lim;
		if (k == rk) return ;
		rk = k, lim = 1 << k;
		for (int i = 1; i <= lim; ++i) {
		int j = rev[i - 1], t = k - 1;
		while (t >= 0 && ((j >> t) & 1)) {
		j ^= 1 << t; --t;
		}
		if (t >= 0) {
		j ^= 1 << t; --t;
		}
		rev[i] = j;
		}
}
void fft(cmplx *a, int k) {
		ReserveBits(k);
		int n = 1 << k;
		for (int i = 0 ; i < n; ++i)
		if (rev[i] > i) swap(a[i], a[rev[i]]);
		for (int l = 2, m = 1; l <= n; l += l, m += m) {
		if (w[m].get_real() == 0 && w[m].get_img() == 0) {
		double angle = M_PI / m;
		cmplx ww(cos(angle), sin(angle));
		if (m > 1) {
		for (int j = 0; j < m; ++j) {
		if (j & 1) w[m + j] = w[(m + j) / 2] * ww;
		else w[m + j] = w[(m + j) / 2];
		}
		}
		else w[m] = cmplx(1, 0);
		}
		for (int i = 0; i < n; i += l) {
		for (int j = 0; j < m; ++j) {
		cmplx v = a[i + j], u = a[i + j + m] * w[m + j];
		a[i + j] = v + u;
		a[i + j + m] = v - u;
		}
		}
		}
}
vector<long long> mul(const vector<long long>& a, const vector<long long>& b) {
		int k = 1;
		while ((1 << k) < (a.size() + b.size())) ++k;
		int n = (1 << k);
		for (int i = 0; i < n; ++i) f1[i] = cmplx(0, 0);
		for (int i = 0; i < a.size(); ++i) f1[i] = f1[i] + cmplx(a[i], 0);
		for (int i = 0; i < b.size(); ++i) f1[i] = f1[i] + cmplx(0, b[i]);
		fft(f1, k);
		for (int i = 0; i <= n / 2; ++i) {
		cmplx p = f1[i] + f1[(n - i) % n].conj();
		cmplx _q = f1[(n - i) % n] - f1[i].conj();
		cmplx q(_q.get_img(), _q.get_real());
		f1[i] = (p * q) * 0.25;
		if (i > 0) f1[(n - i)] = f1[i].conj();
		}
		for (int i = 0; i < n; ++i) f1[i] = f1[i].conj();
		fft(f1, k);
		vector<long long> ans(a.size() + b.size() - 1);
		for (int i = 0; i < ans.size(); ++i) {
		ans[i] = (long long) (f1[i].get_real() / n + 0.5) % prime;
		}
		return ans;
}
};
//basic fft code i.e recursive
vector<base> omega;
int FFT_N;
inline void init_fft(long long n)
{
		FFT_N  = n;
		omega.resize(n);
		double angle = 2 * PI / n;

		for (long long i = 0; i < n; i++)
		omega[i] = base( cos(i * angle), sin(i * angle));
}
inline void fft (vector<base> & a)
```

```
{
        int n = (int) a.size();
        if (n == 1)  return;
        int half = n >> 1;

        vector<base> even (half),  odd (half);
        for (int i = 0, j = 0; i < n; i += 2, ++j)
        {
        even[j] = a[i];
        odd[j] = a[i + 1];
        }
        fft (even), fft (odd); // compute the fft of odd part and even part and combine using this
formula
        //based on the recurrence that A(x) = A{even}(x^2) + (x * A{odd}(x^2));
        for (int i = 0, fact = FFT_N / n; i < half; ++i)
        {
        base twiddle =  odd[i] * omega[i * fact] ;
        a[i] =  even[i] + twiddle;
        a[i + half] = even[i] - twiddle;
        }
}
inline void multiply (const vector<int> & a, const vector<int> & b, vector<int> & res)
{
        vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
        long long n = 1;
        while (n < 2 * max (a.size(), b.size()))  n <<= 1;
        fa.resize (n),  fb.resize (n);

        init_fft(n);
        // step 1 : Convert A(x) and B(x) from coefficient form to point value form. (FFT)
        fft (fa),  fft (fb);
        // step 2: Now do the O(n) convolution in point value form to obtain C(x) in point value
form,
        //  i.e. basically C(x) =A(x) * B(x) in point value form.
        for (size_t i = 0; i < n; ++i)
        fa[i] = conj( fa[i] * fb[i]);
        fft (fa);
        res.resize (n);
        // step3 : Now convert C(x) from point value from to coefficient form (Inverse FFT).
        for (size_t i = 0; i < n; ++i)
        {
        res[i] = (int) (fa[i].real() / n + 0.5);
        res[i] %= prime;
        }
}
// multinomial theorm solution
vector <long long > go(int lo , int hi)
{
        vector<long long> ret;
        if (lo == hi)
        {
        ret.resize(deg[lo] + 1);
        for (int i = 0 ; i <= deg[lo]; i ++)
        ret[i] = 1;
        return ret;
        }
        vector <long long> a, b;
        int mid = (lo + hi) / 2;
        a = go(lo, mid);
        b = go(mid + 1, hi);
        ret = FFT::mul(a, b);
        return ret;
}
```

Miller Rabin Primality Testing (Count No: of divisors)

```
// counting the number of divisors of a given number in O(n^(1/3))
const int N = 1e6 + 1;
bool pr[N];
vector<long long>primes;
void pre() {
        //computing primes till 10^6
        pr[0] = 1, pr[1] = 1;
        for (ll i = 2; i * i < N ; i ++) {
        if (!pr[i]) {
        // primes.push_back(i);
        for (ll j = i * i; j < N; j += i) {
        pr[j] = 1;
        }
        }
        }
        for (int i = 0 ; i < N ; i++) {
        if (!pr[i])
        primes.push_back(i);
        }
        return;
}
ll mulmod(ll a, ll b, ll mod)
{
        ll x = 0, y = a % mod;
        while (b > 0)
        {
        if (b % 2 == 1)
        x = (x + y) % mod;

        y = (y  + y) % mod;
        b /= 2;
        }
        return x % mod;
}
ll modulo(ll base, ll exponent, ll mod)
```

```
{
    ll x = 1;
    ll y = base;
    while (exponent > 0)
    {
    if (exponent % 2 == 1)
    x = mulmod(x, y, mod);
    y = mulmod(y, y, mod);
    exponent /= 2;
    }
    return x % mod;
}
bool Miller(ll p, int iteration)
{
    if (p < 2)
    return false;

    if (p == 2)
    return true;
    if (p != 2 && p % 2 == 0)
    return false;

    ll s = p - 1;
    while (s % 2 == 0)
    s /= 2;

    for (int i = 0; i < iteration; i++)
    {
    ll a = (rand()) % (p - 1) + 1;
    ll temp = s;
    ll mod = modulo(a, temp, p);
    if (mod == 1 || mod == -1)
    continue;
    while (temp != p - 1 && mod != 1 && mod != p - 1)
    {
    mod = mulmod(mod, mod, p);
    temp *= 2;
    }
    if (mod != p - 1 && temp % 2 == 0)
    return false;
    }
    return true;
}
int main() {
    //freopen("../input.txt", "r", stdin);
    pre();
    long long n;
    cin >> n;
    long long res = 1 ;
    for (long long p : primes) {
    if ((p * p * p) > n) {
    break;
    }
    long long exp = 1;
    while (n % p == 0) {
    n /= p;
    exp ++;
    }
    res *= exp;
    }
    long long x = sqrt(n);
    if (Miller(n , 100))
    res = (res * 2);
    else if ((x * x == n && Miller(x, 100)))
    res = (res * 3);
    else if (n != 1)
    res = (res * 4);
    cout << res << endl;
    return 0;
}
```

Chinese Remainder Theorem

```
#include <iostream>
using namespace std;
// returns x where (a * x) % b == 1
int mul_inv(int a, int b)
{
    int b0 = b, t, q;
    int x0 = 0, x1 = 1;
    if (b == 1) return 1;
    while (a > 1) {
    q = a / b;
    t = b, b = a % b, a = t;
    t = x0, x0 = x1 - q * x0, x1 = t;
    }
    if (x1 < 0) x1 += b0;
    return x1;
}

int chinese_remainder(int *n, int *a, int len)
{
    int p, i, prod = 1, sum = 0;

    for (i = 0; i < len; i++) prod *= n[i];

    for (i = 0; i < len; i++) {
    p = prod / n[i];
    sum += a[i] * mul_inv(p, n[i]) * p;
    }

    return sum % prod;
```

```
}

int main(void)
{
	// X % N[i] = A[i]
	// _gcd(N[i],N[j]) = 1  for all i not equal to j mus t condition then only it is valid
	int n[] = { 3, 5, 7 };
	int a[] = { 2, 3, 2 };
	// to find the remainder with N[0] * N[1] * N[2] .... using this information
	printf("%d\n", chinese_remainder(n, a, sizeof(n) / sizeof(n[0])));
	return 0;
}
```

Matrix Exponentiation

```
/*------------MATRIX EXPONENTIATION IN O(logn)-----------------*/

struct matrix
{
   int M[2][2];
   matrix()
   {
	M[0][0] = M[0][1] = M[1][0] = M[1][1] = 0;
   }
   matrix(int a)
   {
	M[0][0] = M[1][1] = a;
	M[1][0] = M[0][1] = 0;
   }
};
inline matrix operator+(matrix A, matrix B)   //addition operator on two matrix
{
   matrix C;
   for (int i = 0; i < 2; i++)
   {
	for (int j = 0; j < 2; j++)
	{
		C.M[i][j] = ((ll)A.M[i][j] + (ll)B.M[i][j]) % mod;
	}
   }
   return C;
}
inline matrix operator*(matrix A, matrix B)  //multiplication operator on matrix
{
   matrix C;
   for (int i = 0; i < 2; i++)
   {
	for (int j = 0; j < 2; j++)
	{
		for (int k = 0; k < 2; k++)
		{
			C.M[i][j] += ((ll)A.M[i][k] * (ll)B.M[k][j]) % mod;
		}
	}
   }
   return C;
}

inline matrix matpow(matrix& A, ll exp)  // matrix exponentiation
{
   matrix X = identity;  // here identity is identity-matrix
   // i.e -> identity.M[0][0] = identity.M[1][1] = 1;
   matrix Y = A;
   while (exp)
   {
	if (exp & 1ll)
	{
		X = X * Y;
	}
	Y = Y * Y;
	exp >>= 1ll;
   }
   return X;
}
```

Euler's Totient Function

```
/*-------EULER TOTIENT FUNCTION FOR SINGLE NUMBER IN SQRT(N)--------*/
//    phi(n) = number of positive integers less than n which are coprime to n
//    i.e count of x, 1 <= x <= n such that gcd(x, n) = 1
//Time Complexity- O(sqrt(N));

ll phi(long long x)
{
   ll ret = 1, i, pow;
   for (int i = 2; x != 1; i++)
   {
	pow = 1;
	if (i > sqrt(x))break;
	while (!(x % i))
	{
		x /= i;
		pow *= i;
	}
	ret *= (pow - (pow / i));
   }
   if (x != 1)
	ret *= (x - 1);
   return ret;
}
/*---------------EULER TOTIENT FUNCTION---------------------*/
//    phi(n) = number of positive integers less than n which are coprime to n
```

```
//    i.e count of x, 1 <= x <= n such that gcd(x, n) = 1

inline void phi()
{
   for (int i = 1; i < SIZE; i++)
   {
      phi[i] = i;
   }
   prime[0] = prime[1] = 1;
   for (int i = 2; i < SIZE; i++)
   {
      if (prime[i] == 0)
      {
         phi[i] = i - 1;
         for (int j = 2 * i; j < SIZE; j += i)
         {
            prime[j] = 1;
            phi[j] = (phi[j] * (i - 1)) % mod;
            phi[j] = (phi[j] / i);
         }
      }
   }
}
```

Gauss Jordan

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
// (1) solving systems of linear equations (AX=B)
// (2) inverting matrices (AX=I)
// (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT: a[][] = an nxn matrix
// b[][] = an nxm matrix
//
// OUTPUT: X = an nxm matrix (stored in b[][])
// A^{-1} = an nxn matrix (stored in a[][])
// returns determinant of a[][]

const double EPS = 1e-10;
typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
T GaussJordan(VVT &a, VVT &b) {
   const int n = a.size();
   const int m = b[0].size();
   VI irow(n), icol(n), ipiv(n);
   T det = 1;
   for (int i = 0; i < n; i++) {
      int pj = -1, pk = -1;
      for (int j = 0; j < n; j++) if (!ipiv[j])
         for (int k = 0; k < n; k++) if (!ipiv[k])
            if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j;
pk = k; }
      if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
      ipiv[pk]++;
      swap(a[pj], a[pk]);
      swap(b[pj], b[pk]);
      if (pj != pk) det *= -1;
      irow[i] = pj;
      icol[i] = pk;
      T c = 1.0 / a[pk][pk];
      det *= a[pk][pk];
      a[pk][pk] = 1.0;
      for (int p = 0; p < n; p++) a[pk][p] *= c;
      for (int p = 0; p < m; p++) b[pk][p] *= c;
      for (int p = 0; p < n; p++) if (p != pk) {
         c = a[p][pk];
         a[p][pk] = 0;
         for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
         for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
      }
   }
   for (int p = n - 1; p >= 0; p--) if (irow[p] != icol[p]) {
      for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
   }
   return det;
}
int main() {
   const int n = 4;
   const int m = 2;
   double A[n][n] = { {1, 2, 3, 4}, {1, 0, 1, 0}, {5, 3, 2, 4}, {6, 1, 4, 6} };
   double B[n][m] = { {1, 2}, {4, 3}, {5, 6}, {8, 7} };
   VVT a(n), b(n);
   for (int i = 0; i < n; i++) {
      a[i] = VT(A[i], A[i] + n);
      b[i] = VT(B[i], B[i] + m);
   }
   double det = GaussJordan(a, b);
// expected: 60
   cout << "Determinant: " << det << endl;
// expected: -0.233333 0.166667 0.133333 0.0666667
// 0.166667 0.166667 0.333333 -0.333333
// 0.233333 0.833333 -0.133333 -0.0666667
// 0.05 -0.75 -0.1 0.2
   cout << "Inverse: " << endl;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
```

```
                cout << a[i][j] << ' ';
            cout << endl;
    }
// expected: 1.63333 1.3
// -0.166667 0.5
// 2.36667 1.7
// -1.85 -1.35
    cout << "Solution: " << endl;
    for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++)
                cout << b[i][j] << ' ';
            cout << endl;
    }
}
```

Persistent Segment Tree

```
//    Persistent Segment Tree
//    Query : O(logn)
//    Space : O(n * logn)
//    Build : O(n * logn)
int root[MAX], lft[20 * MAX];
int rht[20 * MAX], tree[20 * MAX];
int next_index = 0, n;

int insert(int id, int l, int r, int i)
{
    int ID = ++next_index;
    if (l == r)
    {
            tree[ID] = tree[id] + 1;
            return ID;
    }
    lft[ID] = lft[id];
    rht[ID] = rht[id];
    int mid = (l + r) >> 1;
    if (i <= mid)
            lft[ID] = insert(lft[id], l, mid, i);
    else
            rht[ID] = insert(rht[id], mid + 1, r, i);
    tree[ID] = tree[lft[ID]] + tree[rht[ID]];
    return ID;
}

int query(int r1, int r2, int l, int r, int i)
{
    if (l == r)
            return l;
    int cnt = tree[lft[r2]] - tree[lft[r1]]; // Calculate Value of left subtree
    int mid = (l + r) >> 1;
    if (cnt >= i)    //    Check if ans lies in left subtree
            return query(lft[r1], lft[r2], l, mid, i);
    return query(rht[r1], rht[r2], mid + 1, r, i - cnt);
}

// Main Function
for (int i = 1; i <= n; i++)
    root[i] = insert(root[i - 1], 1, n, val[i]);
```

Treap

```
const int MAXN = 1e5 + 10;
using namespace std;

struct Node {
        bool rev;
        Node *l, *r;
        int val, subtree, priority;

        inline Node() {
        l = r = 0;
        }

        inline Node(long long v) {
        l = r = 0;
        priority = rand();
        rev = false, subtree = 1, val = v;
        }

        inline void update() {
        subtree = 1;
        if (l) subtree += l->subtree;
        if (r) subtree += r->subtree;
        }
} nodes[MAXN]; /// Maximum number of nodes in treap

struct Treap {
        int idx;
        struct Node* root;

        inline void push(Node* cur) {
        if (cur && cur->rev) {
        cur->rev = false;
        if (cur->l) cur->l->rev ^= true;
        if (cur->r) cur->r->rev ^= true;
        swap(cur->l, cur->r);
        }
        }

        inline void merge(Node* &cur, Node* l, Node* r) {
        push(l), push(r);
        if (!l || !r) cur = l ? l : r;
```

```
else if (l->priority > r->priority) merge(l->r, l->r, r), cur = l;
else merge(r->l, l, r->l), cur = r;
if (cur) cur->update();
}

inline void split(Node* cur, Node* &l, Node* &r, int key, int counter = 0) {
if (!cur) {
l = r = 0;
return;
}

push(cur);
int cur_key = counter + (cur->l ? cur->l->subtree : 0);
if (key <= cur_key) split(cur->l, l, cur->l, key, counter), r = cur;
else split(cur->r, cur->r, r, key, cur_key + 1), l = cur;
if (cur) cur->update();
}

inline void insert(int i, int v) {
nodes[idx] = Node(v);
merge(root, root, &nodes[idx++]);
}

inline void update(int a, int b, int c) {
Node *l, *r, *m;
split(root, l, r, a);
split(r, m, r, b);
merge(r, l, r);
split(r, l, r, c);
m->rev ^= true;
merge(m, m, r);
merge(root, l, m);
}

Treap() {
srand(time(0));
idx = 0, root = 0;
}

void dfs(Node* cur) {
if (!cur) return;
push(cur);
dfs(cur->l);
printf("%d ", cur->val);
dfs(cur->r);
}

void dfs() {
dfs(root);
}
};

int main() {
        int n, m, i, j, a, b, c;
#ifdef LOCAL_PROJECT
        freopen("../input.txt", "r", stdin);
        //freopen("../output.txt","w",stdout);
#endif

        while (scanf("%d %d", &n, &m) != EOF) {
        Treap T = Treap();
        for (i = 1; i <= n; i++) T.insert(i, i);

        while (m--) {
        scanf("%d %d %d", &a, &b, &c);
        T.update(a, b, c);
        }

        T.dfs();
        puts("");
        }

        return 0;
}
```

Suffix Array (n * logn * logn)

```
//   Suffix Array
//    Build : O(n * logn * logn)
//   LCP : O(logn)
struct node
{
   int x[2];
   int pos;
} L[MAX];
char s[MAX];
int n, m, l, sarray[12][MAX];
pii ar[MAX];

bool comp(node a, node b)
{
   if (a.x[0] == b.x[0])
           return (a.x[1] < b.x[1]);
   else
           return (a.x[0] < b.x[0]);
}

void suffixarray()
{
   for (int i = 0; i < n; i++)
           sarray[0][i] = s[i] - 'A';
```

```
    int cnt;
    for (m = 1, cnt = 1; cnt >> 1 < n; m++, cnt <<= 1)
    {
        for (int i = 0; i < n; i++)
        {
            L[i].x[0] = sarray[m - 1][i];
            if (i + cnt < n)
                L[i].x[1] = sarray[m - 1][i + cnt];
            else
                L[i].x[1] = -1;
            L[i].pos = i;
        }
        sort(L, L + n, comp);
        for (int i = 0; i < n; i++)
        {
            if (i == 0)
            {
                sarray[m][L[0].pos] = 0;
                continue;
            }
            if (L[i].x[0] == L[i - 1].x[0] && L[i].x[1] == L[i - 1].x[1])
                sarray[m][L[i].pos] = sarray[m][L[i - 1].pos];
            else
                sarray[m][L[i].pos] = sarray[m][L[i - 1].pos] + 1;
        }
    }
}

int lcp(int x, int y)
{
    if (x == y)
        return n - x;
    int ans = 0;
    for (int i = m - 1; i >= 0 && x < n && y < n; i--)
    {
        if (sarray[i][x] == sarray[i][y])
        {
            ans += (1 << i);
            x += (1 << i);
            y += (1 << i);
        }
    }

    return ans;
}
```

Suffix Array (n * logn)

```
//Building Suffix Array
//Time Complexity- O(NlogN)
#define SIZE 1000010
int n, gap = 0, c;
int bucket[SIZE], temp[SIZE], lcp[SIZE], LCP[SIZE][22], start_idx[SIZE];
string s, st;
std::vector<string> vec;
//Usage:
// Fill str with the characters of the string.
// Call SA(n), where n is the length of the string stored in str.

//Output:
// pos = The suffix array. Contains the n suffixes of str sorted in lexicographical order.
//        Each suffix is represented as a single integer (the position of str where it starts).
// bucket = The inverse of the suffix array. bucket[i] = the index of the suffix str[i..n)
//        in the pos array. (In other words, pos[i] = k <==> bucket[k] = i)
//        With this array, you can compare two suffixes in O(1): Suffix str[i..n) is smaller
//        than str[j..n) if and only if ran[i] < ran[j]
//lcp[i] = length of the longest common prefix of suffix pos[i] and suffix pos[i-1]
// lcp[0] = 0

struct node {
    int idx;  // Suffix starts at idx, i.e. it's str[ idx .. L-1 ]
    bool operator<(const node& sfx) const
    {
        // Compares two suffixes based on their first 2H symbols,
        // assuming we know the result for H symbols.

        if (gap == 0)
            return s[idx] < s[sfx.idx];
        else if (bucket[idx] == bucket[sfx.idx])
            return bucket[idx + gap] < bucket[sfx.idx + gap];
        else
            return bucket[idx] < bucket[sfx.idx];
    }
    bool operator==(const node& sfx) const
    {
        return !(*this < sfx) && !(sfx < *this);
    }
} pos[SIZE];

int updatebucket()
{
    int start = 0, id = 0, c = 0;
    for (int i = 0; i < n; i++)
    {
        /*
            If Pos[i] is not equal to Pos[i-1], a new bucket has started.
        */

        if (i != 0 && !(pos[i] == pos[i - 1]))
        {
            start = i;
            id++;
```

```
        }
        if (start != i)   // if there is bucket with size larger than 1, we should continue ...
            c = 1;
        temp[pos[i].idx] = id;  // Bucket for suffix starting at Pos[i].idx is id ...
    }
    memcpy(bucket, temp, 4 * n);
    return c;
}

void buildSA()
{
    for (int i = 0; i < n; i++)
    {
        pos[i].idx = i;
    }// gap == 0, Sort based on first Character.
    sort(pos, pos + n);
    // Create initial buckets
    c = updatebucket();
    for (gap = 1; c; gap *= 2)
    {
        // Sort based on first 2*gap symbols, assuming that we have sorted based
        //on first gap character
        sort(pos, pos + n);
        // Update Buckets based on first 2*gap symbols
        c = updatebucket();
    }
}

void buildLCP()
{
    lcp[0] = 0;
    for (int i = 0, h = 0; i < n; ++i)
    {
        if (bucket[i] > 0) {
            int j = pos[bucket[i] - 1].idx;
            while (i + h < n && j + h < n && s[i + h] == s[j + h])
                h++;
            lcp[bucket[i]] = h;
            if (h > 0)
                h--;
        }
    }
    for (int i = 0; i < n; ++i)
        LCP[i][0] = lcp[i];

    for (int j = 1; (1 << j) <= n; ++j)
    {
        for (int i = 0; i + (1 << j) - 1 < n; ++i)
        {
            if (LCP[i][j - 1] <= LCP[i + ( 1 << (j - 1) )][j - 1])
                LCP[i][j] = LCP[i][j - 1];
            else
                LCP[i][j] = LCP[i + (1 << (j - 1))][j - 1];
        }
    }
}
int get_lcp(int x, int y)
{
    if (x == y)
        return n - pos[x].idx;
    if (x > y)
        swap(x, y);
    int log = 0;
    while ((1 << log) <= (y - x)) ++log;
    --log;
    return min(LCP[x + 1][log], LCP[y - (1 << log) + 1][log]);
}


int main()
{
    cin >> s;
    n = s.size();
    buildSA();
    buildLCP();
    for (int i = 0; i < n; i++)
    {
        cout << pos[i].idx << endl;
    }
    return 0;
}
```

Binary Indexed Tree

```
void update(int i, int v)    //    Point Update
{
    for(; i <= n; i += (i & -i))
        bit[i] += v;
    return;
}

int query(int i)    //    Query of 1 <= j <= i
{
    int sum = 0;
    for(; i > 0; i -= (i & -i))
        sum += bit[i];
    return sum;
}

//    Binary Indexed Tree
//    Range Update, add v to each element in A[a..b]
//    Range Query, get sum of each element in A[a..b]
```

```
update(ft, p, v):
  for (; p <= N; p += p&(-p))
        ft[p] += v

// Add v to A[a...b]
update(a, b, v):
  update(B1, a, v)
  update(B1, b + 1, -v)
  update(B2, a, v * (a-1))
  update(B2, b + 1, -v * b)

query(ft, b):
  sum = 0
  for(; b > 0; b -= b&(-b))
        sum += ft[b]
  return sum

// Return sum A[1...b]
query(b):
  return query(B1, b) * b - query(B2, b)

// Return sum A[a...b]
query(a, b):
return query(b) - query(a-1)
```

DP Optimizations

Name	**Original Recurrence**	**Sufficient Condition**	**Complexity**	
			Original	**Optimized**
Convex Hull 1	*dp[i] = $\min_{j<i}$ {dp[j] + b[j]*a[i]}*	*b[j] b[j + 1]* *a[i] a[i + 1]*	$O(n^2)$	$O(n)$
Convex Hull 2	*dp[i][j] = $\min_{k<j}$ {dp[i-1][k] + b[k]*a[j]}*	*b[k] b[k + 1]* *a[j] a[j + 1]*	$O(kn^2)$	$O(kn)$
Divide & Conquer	*dp[i][j] = $\min_{k<j}$ {dp[i-1][k] + C[k][j]}*	*A[i][j] A[i][j + 1]*	$O(kn^2)$	$O(knlogn)$
Knuth Optimization	*dp[i][j] = $\min_{i<k<j}$ {dp[i][k] + dp[k][j]} + C[i][j]*	*A[i][j - 1]* *A[i][j]* *A[i + 1][j]*	$O(n^3)$	$O(n^2)$

Notes:

- *A[i][j] — the smallest k that gives optimal answer, for example in dp[i][j] = dp[i - 1][k] + C[k][j]*
- *C[i][j] — some given cost function*
- *We can generalize a bit in the following way: dp[i] = $\min_{j<i}$ {F[j] + b[j] * a[i]}, where F[j] is computed from dp[j] in constant time.*
- *It looks like **Convex Hull Optimization 2** is a special case of **Divide and Conquer Optimization**.*
- *It is claimed (in the references) that **Knuth Optimization** is applicable if C[i][j] satisfies the following 2 conditions:*
- ***quadrangle inequality**:* $C[a][c] + C[b][d] \le C[a][d] + C[b][c],\ a \le b \le c \le d$
- ***monotonicity**:* $C[b][c] \le C[a][d],\ a \le b \le c \le d$

Divide & Conquer Optimization

```
//  Divide & Conquer optimization
//  Time Complexity : O(K * n * logn)
int n, K;
ll dp[505][10013];

void solve(int i, int j1, int j2, int k1, int k2)
{
   if (j1 > j2 || k1 > k2)
          return;

   int mid = (j1 + j2) / 2;
   int temp = -1;
   dp[i][mid] = mod;
   for (int k = k1; k <= min(mid, k2); k++)
   {
          ll v = (mid - k) * (pre[mid] - pre[k]);
          if (dp[i][mid] > dp[i - 1][k] + v)
          {
                 dp[i][mid] = dp[i - 1][k] + v;
                 temp = k;
          }
   }
   solve(i, j1, mid - 1, k1, temp);
   solve(i, mid + 1, j2, temp, k2);
}

int main()
{
   ind(n);
   ind(K);
   for (int i = 1; i <= n; i++)
   {
```

```
        dp[1][i] = // base case
    }
    for (int i = 2; i <= K; i++)
    {
        solve(i, i, n, i, n);
    }
    pr(dp[K][n]);
    return 0;
}
```

Knuth Optimization

```
// Knuth Optimization
// Complexity - O(n*n)

for (int s = 0; s <= k; s++)                  //s - length(size) of substring
    for (int L = 0; L + s <= k; L++) {        //L - left point
        int R = L + s;                        //R - right point
        if (s < 2) {
            res[L][R] = 0;                    //DP base - nothing to break
            mid[L][R] = l;                    //mid is equal to left border
            continue;
        }
        int mleft = mid[L][R - 1];            //Knuth's trick: getting bounds on M
        int mright = mid[L + 1][R];
        res[L][R] = 1000000000000000000LL;
        for (int M = mleft; M <= mright; M++) {   //iterating for M in the bounds only
            int64 tres = res[L][M] + res[M][R] + (x[R] - x[L]);
            if (res[L][R] > tres) {           //relax current solution
                res[L][R] = tres;
                mid[L][R] = M;
            }
        }
    }
int64 answer = res[0][k];
```

Convex Hull Optimization 1 & 2

```
// used for recurrences such as
// convex hull optimisation
// dp[i] = min(dp[j] + a[i]*b[j]) for all j < i such as b[j] >=b[j + 1] and a[i] <= a[i + 1]
// mf[i] minimumm fuction till now by considering i points till now
// O(n^2) --->  O(n) using this convex hull optimisation
double slope(int x , int y)
{
    return (1.0 * (dp[y] - dp[x]) / (1.0 * (b[y] - b[x])));
}
mf[r ++] = 1;
for (int i = 2 ; i <= n ; i ++)
{

    while (r - l >= 2 && slope(mf[l], mf[l + 1]) > -a[i]) // finding the minimum point
        l ++;
    dp[i] = dp[mf[l]] + 1LL * (b[mf[l]] * a[i]); // updating my dp with the minimum found function of this range
    //i.e from considering functions from lines [1..i-1]
    while (r - l >= 2 && slope(mf[r - 1], mf[r - 2]) < slope(mf[r - 1], i))r--;//removing the unnecesary points form
    // these points will not be the part of the convex  hull
    mf[r++] = i; // index of the minimum found function till now
}

// second optimisation of convex hull
// same as convex hull first optimsation but for 2d rec
// dp[i][j] = min(dp[i-1][k] + b[k]*a[j]) for all k < j provided b[k] >= b[k + 1] && a[j] <= a[j + 1]
```

Geometry

```
// C++ routines for computational geometry.
#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>
using namespace std;
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 ); }
};

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 << ")";
}

// 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) {
```

```
  return PT(p.x * cos(t) - p.y * sin(t), p.x * sin(t) + p.y * cos(t));
}

// 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 (fabs(r) < EPS) return a;
  r = dot(c - a, b - a) / r;
  if (r < 0) return a;
  if (r > 1) 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 fabs(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 fabs(cross(b - a, c - d)) < EPS;
}
bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
        && fabs(cross(a - b, a - c)) < EPS
        && fabs(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) > 0 && dot(d - a, d - b) > 0 && dot(c - b, d - b) > 0)
            return false;
        return true;
  }
  if (cross(d - a, b - a) * cross(c - a, b - a) > 0) return false;
  if (cross(a - c, d - c) * cross(b - c, d - c) > 0) 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);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b = (a + b) / 2;
  c = (a + c) / 2;
  return ComputeLineIntersection(b, b + RotateCW90(a - b), c, c + RotateCW90(a - c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++) {
        int j = (i + 1) % p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
              p[j].y <= q.y && q.y < p[i].y) &&
              q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
              c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i + 1) % p.size()], q), q) < EPS)
              return true;
  return false;

}
// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
```

```
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
   vector<PT> ret;
   b = b - a;
   a = a - c;
   double A = dot(b, b);
   double B = dot(a, b);
   double C = dot(a, a) - r * r;
   double D = B * B - A * C;
   if (D < -EPS) return ret;
   ret.push_back(c + a + b * (-B + sqrt(D + EPS)) / A);
   if (D > EPS)
          ret.push_back(c + a + b * (-B - sqrt(D)) / A);
   return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
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 fabs(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;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
   for (int i = 0; i < p.size(); i++) {
          for (int k = i + 1; k < p.size(); k++) {
                 int j = (i + 1) % p.size();
                 int l = (k + 1) % p.size();
                 if (i == l || j == k) continue;
                 if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                        return false;
          }
   }
   return true;
}

int main() {
#ifndef ONLINE_JUDGE
   freopen("input.txt", "r", stdin);
#endif
// 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 1 1 0
   cerr << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(3, 1), PT(-1, 3)) << " "
          << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(4, 3), PT(0, 5)) << " "
          << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(2, -1), PT(-2, 1)) << " "
          << SegmentsIntersect(PT(0, 0), PT(2, 4), PT(5, 5), PT(1, 7)) << endl;
// expected: (1,2)
   cerr << ComputeLineIntersection(PT(0, 0), PT(2, 4), PT(3, 1), PT(-1, 3)) << endl;
// expected: (1,1)
```

```
    cerr << ComputeCircleCenter(PT(-3, 4), PT(6, 1), PT(4, 5)) << endl;
    vector<PT> v;
    v.push_back(PT(0, 0));
    v.push_back(PT(5, 0));
    v.push_back(PT(5, 5));
    v.push_back(PT(0, 5));
// expected: 1 1 1 0 0
    cerr << PointInPolygon(v, PT(2, 2)) << " "
         << PointInPolygon(v, PT(2, 0)) << " "
         << PointInPolygon(v, PT(0, 2)) << " "
         << PointInPolygon(v, PT(5, 2)) << " "
         << PointInPolygon(v, PT(2, 5)) << endl;
// expected: 0 1 1 1 1
    cerr << PointOnPolygon(v, PT(2, 2)) << " "
         << PointOnPolygon(v, PT(2, 0)) << " "
         << PointOnPolygon(v, PT(0, 2)) << " "
         << PointOnPolygon(v, PT(5, 2)) << " "
         << PointOnPolygon(v, PT(2, 5)) << endl;
// expected: (1,6)
// (5,4) (4,5)
// blank line
// (4,5) (5,4)
// blank line
// (4,5) (5,4)
    vector<PT> u = CircleLineIntersection(PT(0, 6), PT(2, 6), PT(1, 1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleLineIntersection(PT(0, 9), PT(9, 0), PT(1, 1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1, 1), PT(10, 10), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1, 1), PT(8, 8), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1, 1), PT(4.5, 4.5), 10, sqrt(2.0) / 2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1, 1), PT(4.5, 4.5), 5, sqrt(2.0) / 2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
// area should be 5.0
// centroid should be (1.1666666, 1.166666)
    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;
    return 0;
}
```

Convex Hull - Graham Scan

```
//   Convex Hull - Graham Scan
//   Time Complexity : O(n * logn)
struct point
{
    int x, y, pos;
}P[MAX], p0;
int n, m;
vector<point> hull;

int distance(point p, point q)
{
    int X = p.x - q.x;
    int Y = p.y - q.y;
    int d = X * X + Y * Y;
    return d;
}

int orientation(point p, point q, point r)
{
        int val = (q.y - p.y) * (r.x - q.x) -
        (q.x - p.x) * (r.y - q.y);

        if (val == 0)
         return 0;  // colinear
        return ((val > 0)? 1: 2); // clock or counterclock wise
}

bool comp(point a, point b)
{
    int o = orientation(p0, a, b);
    if (o != 0)
        return ((o == 1) ? 0 : 1);
    int d1 = distance(p0, a);
    int d2 = distance(p0, b);
    if (d1 != d2)
        return ((d1 > d2) ? 0 : 1);
    return (a.pos > b.pos);
}

double convexhull()
{
    hull.clear();
    m = 1;
    if (n <= 0)
        return 0.0;
    int id = 0, miny = P[0].y;
    for (int i = 1; i < n; i++)
    {
        if (P[i].y < miny)
            id = i, miny = P[i].y;
        else if (P[i].y == miny && P[i].x < P[id].x)
            id = i;
    }
    swap(P[0], P[id]);
```

```
    p0 = P[0];
    sort(P + 1, P + n, comp);
    for (int i = 1; i < n; i++)
    {
        while (i < n - 1 && orientation(p0, P[i], P[i + 1]) == 0)
            i++;
        P[m++] = P[i];
    }
    hull.push_back(p0);
    if (m == 2)
    {
        hull.push_back(P[1]);
        return (2.0 * sqrt(distance(P[0], P[1])));
    }
    if (m < 3)
        return 0.0;
    hull.push_back(P[1]);
    hull.push_back(P[2]);

    for (int i = 3; i < m; i++)
    {
        int sz = hull.size() - 1;
        while (sz > 0 && orientation(hull[sz - 1], hull[sz], P[i]) != 2)
        {
            hull.pop_back();
            sz--;
        }
        hull.push_back(P[i]);
    }
    double d = 0.0;
    for (int i = 1; i < hull.size(); i++)
        d += sqrt(distance(hull[i], hull[i - 1]));
    d += sqrt(distance(hull[0], hull.back()));
    return d;
}
```

Offline Bridge Searching

```
//  Bridge Searching
//  Time Complexity : O(V + E)
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
    used[v] = true;
    tin[v] = fup[v] = timer++;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (to == p)  continue;
        if (used[to])
            fup[v] = min (fup[v], tin[to]);
        else {
            dfs (to, v);
            fup[v] = min (fup[v], fup[to]);
            if (fup[to] > tin[v])
                IS_BRIDGE(v,to);
        }
    }
}

void find_bridges() {
    timer = 0;
    for (int i=0; i<n; ++i)
        used[i] = false;
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs (i);
```

Articulation Point

```
//  Articulation Point
/*
    In a graph, a vertex is called an articulation point if
    removing it and all the edges associated with it results in
    the increase of the number of connected components in the graph.
*/
//  Time Complexity : O(V + E)
int n, m, t = -1, tin[MAX], fup[MAX];
vector<int> v[MAX];
bool vis[MAX], isarticulate[MAX];

void dfs(int src, int p)
{
    tin[src] = ++t;
    int x, child = 0;
    fup[src] = tin[src];
    vis[src] = 1;
    for (int i = 0; i < v[src].size(); i++)
    {
        x = v[src][i];
        if (x == p)
            continue;
        if (vis[x])
        {
            fup[src] = min(fup[src], tin[x]);
        }
        else
        {
            dfs(x, src);
            fup[src] = min(fup[src], fup[x]);
```

```
                if (fup[x] >= tin[src] && p != 0)
                        isarticulate[src] = 1;
                child++;
        }
    }
    if (p == 0 && child > 1)
        isarticulate[src] = 1;
}
```

Strongly Connected Component

```
//  Strongly Connected Components
//  Time Complexity : O(V + E)
int n, m;
vector<int> v[5005], vt[5005], scc;
stack<int> s;
bool vis[5005], vis1[5005];

void dfs(int src)   //  Traversal on original graph
{
    vis[src] = 1;
    for (int i = 0; i < v[src].size(); i++)
    {
        if (!vis[v[src][i]])
            dfs(v[src][i]);
    }
    s.push(src);
    return;
}

void dfst(int src)   //  Traversal on reverse graph
{
    vis[src] = 1;
    for (int i = 0; i < vt[src].size(); i++)
    {
        if (!vis[vt[src][i]])
            dfst(vt[src][i]);
    }
    scc.push_back(src);
    return;
}

int main()
{
    int x, y;
    ind(n);
    for (int i = 0; i <= n + 1; i++)
    {
        v[i].clear();
        vt[i].clear();   // Transpose of the graph
    }
    ind(m);
    for (int i = 0; i < m; i++)
    {
        ind(x);
        ind(y);
        v[x].push_back(y);
        vt[y].push_back(x);   //  transpose of the graph
    }
    MS0(vis);
    for (int i = 1; i <= n; i++)
    {
        if (!vis[i])
            dfs(i);
    }
    MS0(vis);
    while (!s.empty())
    {
        int cur = s.top();
        s.pop();
        if (vis[cur])
            continue;
        scc.clear();
        dfst(cur);
        for (int i = 0; i < scc.size(); i++)
        {
            cout << scc[i] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Topological Sorting

```
//  Topological Sort
// Time Complexity : O(V + E)
int n, m, tsort[MAX], sz = 0, indegree[MAX];
vector<int> v[MAX];
bool top_sort()
{
    queue<int> q;
    for(int i = 1; i <= n; i++)
        if(indegree[i] == 0)
            q.push(i);
    int cur;
    while(!q.empty())
    {
        cur = q.top();
        q.pop();
        tsort[sz++] = cur;
        for(int i : v[cur])
```

```
        {
                indegree[i]--;
                if(indegree[i] == 0)
                        q.push(i);
        }
    }

    if(sz < n)
        return 0;
    return 1;
}
```

Parallel Binary Search

```
//    Parallel Binary Search
//    Time Complexity : O(q * logn * logn)
for (int i = 0; i < q; i++)
{
    input(X[0][i]);    input(Y[0][i]);
    input(X[1][i]);    input(Y[1][i]);
    L[i] = 0;    R[i] = mx;
}

int lim = log2(mx) + 1;
for (int i = 0; i < lim; i++)
{
    reset();
    for (int i = 0; i < q; i++)
        if (L[i] != R[i])
            tocheck[(L[i] + R[i]) >> 1].push_back(i);
    for (int i = 0; i <= mx; i++)
    {
        for (auto j : v[i])
            add(j);
        for (int j : tocheck[i])
        {
            if (check(j))
                R[j] = i;
            else
                L[j] = i + 1;
        }
        tocheck[i].clear();
    }
}

for (int i = 0; i < q; i++)
    print(L[i]);
```

MO's With Update

```
/*
        Mo's Algorithm with update operation (N+Q)*pow(N,2/3)
        We divide the blocks into size of pow(N,2/3) instead of pow(N,1/2)
        therefore total complexity is around pow(N,5/3)
*/
//    Compare function for normal MO's Algorithm
bool comp(node a, node b)
{
    if(a.l / block != b.l / block)
        return (a.l / block < b.l / block);
    return a.r <= b.r;
}

bool cmp(query a, query b)
{
        bool res = (B[a.l] < B[b.l] || (B[a.l] == B[b.l] && B[a.r] < B[b.r]));
        res = (res || ((B[a.l] == B[b.l] && B[a.r] == B[b.r]) && a.t < b.t));
    return res;
}

void solve(int node)
{
    if (vis[node])
    {
        // subtract from the ans accordingly
        // update the count
        // reset the vis for this node i.e vis[node]=0;
    }
    else
    {

        // add to the ans accordingly
        // update the count
        // mark the vis for this node i.e vis[node]=1;
    }
}
void change(int node, int col)  // use for update operation
{
    if (vis[node])
    {
        // first call solve(node) func and then subtract
        // the contribution from this node then update the value of node
        // i.e. change the value to col
        // and then again call solve function and add the changed value to the ans
    }
    else
    {
```

```
        // if node is not visited the just update the value at the node
    }
}


int main()
{
    for (int i = 1; i <= n; i++)
    {
        ind(ar[i]);
        previ[i] = ar[i];  //previ[i]=use to keep track of previous value
    }
    for (int i = 1; i <= q; i++)
    {
        // take the input of query and store them
        // store update and query operation differently
        // in update operation store {node_number,update_val,previous_value_at_node};
        // and update the previous_value to updated value;

        // in query operation store {left_value_of_range,
        // right_value_range,update_operation_came_till_now,query_number};
    }
    block_sz = 1500;
    for (int i = 1; i <= tim; i++)
    {
        B[i] = (i - 1 + block_sz) / block_sz;
    }
    sort(Q + 1, Q + 1 + qnum, cmp);

    int cur_L = 0, cur_R = 0, cur_T = 0, lca;
    for (int i = 1; i <= qnum; i++)
    {
        int L = Q[i].l, R = Q[i].r, t = Q[i].t, id = Q[i].id;
        while (cur_T < t) // use for changing the update opeation
        {
            cur_T++;
            change(U[cur_T].node, U[cur_T].col);
        }
        while (cur_T > t)
        {
            change(U[cur_T].node, U[cur_T].pre);
            cur_T--;
        }
        while (cur_L < L)  // use for changing the left_value_of_range
        {
            cur_L++;
            solve(pos[cur_L]);
        }
        while (cur_L > L)
        {
            solve(pos[cur_L]);
            cur_L--;
        }
        while (cur_R < R)// use for changing the right_value_of_range
        {
            cur_R++;
            solve(pos[cur_R]);
        }
        while (cur_R > R)
        {
            solve(pos[cur_R]);
            cur_R--;
        }
        res[id] = ans;
    }
    for (int i = 1; i <= qnum; i++)
        pr(res[i]);
    return 0;

}
```

Heavy Light Decomposition

```
//   Heavy Light Decomposition
//   Query : O(logn * logn)
//   Update : O(logn * logn)
int n, tree[4 * MAX], p[MAX], depth[MAX], m = 0;
int chead[MAX], sz[MAX], otherid[MAX], chain, chainid[MAX];
int baseid[MAX], ar[MAX];
vector<int> v[MAX], cost[MAX], edge[MAX];

void dfs(int src, int par)
{
    depth[src] = depth[par] + 1;
    p[src] = par;
    sz[src] = 1;
    for (int i = 0; i < v[src].size(); i++)
    {
        if (v[src][i] == par)
            continue;
        dfs(v[src][i], src);
        otherid[edge[src][i]] = v[src][i];
        sz[src] += sz[v[src][i]];
    }
}

void hld(int cur, int c, int prev)
{
    if (chead[chain] == -1)
        chead[chain] = cur;
    chainid[cur] = chain;
    baseid[cur] = ++m;
```

```
    ar[m] = c;
    int pos = -1, ncost;
    for (int i = 0; i < v[cur].size(); i++)
    {
        if (v[cur][i] == prev)
            continue;
        if (pos == -1 || sz[v[cur][i]] > sz[pos])
        {
            pos = v[cur][i];
            ncost = cost[cur][i];
        }
    }

    if (pos != -1)
        hld(pos, ncost, cur);

    for (int i = 0; i < v[cur].size(); i++)
    {
        if (v[cur][i] == prev || v[cur][i] == pos)
            continue;
        chain++;
        hld(v[cur][i], cost[cur][i], cur);
    }
}

void build(int id, int l, int r)
{
    if (l == r)
    {
        tree[id] = ar[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(id << 1, l, mid);
    build((id << 1) + 1, mid + 1, r);
    tree[id] = max(tree[id << 1], tree[(id << 1) + 1]);
}

void _update(int id, int l, int r, int i, int val)
{
    if (l == r)
    {
        tree[id] = val;
        return;
    }

    int mid = (l + r) >> 1;
    if (i <= mid)
        _update(id << 1, l, mid, i, val);
    else
        _update((id << 1) + 1, mid + 1, r, i, val);
    tree[id] = max(tree[id << 1], tree[(id << 1) + 1]);
}

void update(int x, int y)
{
    int a = otherid[x];
    _update(1, 1, m, baseid[a], y);
}

int _query(int id, int l, int r, int i, int j)
{
    if (l > j || r < i)
        return -1;
    if (l >= i && r <= j)
        return tree[id];
    int mid = (l + r) >> 1, left, right;
    left = _query(id << 1, l, mid, i, j);
    right = _query((id << 1) + 1, mid + 1, r, i, j);
    return max(left, right);
}

int query(int x, int y)
{
    if (x == y)
        return 0;
    int uc, vc, ans = 0, tmp;
    while (1)
    {
        uc = chainid[x];
        vc = chainid[y];
        if (uc == vc)
        {
            if (x == y)
                break;
            if (depth[y] <= depth[x])
                tmp = _query(1, 1, m, baseid[y] + 1, baseid[x]);
            else
                tmp = _query(1, 1, m, baseid[x] + 1, baseid[y]);
            ans = max(ans, tmp);
            break;
        }
        if (depth[chead[uc]] >= depth[chead[vc]])
        {
            tmp = _query(1, 1, m, baseid[chead[uc]], baseid[x]);
            x = chead[uc];
            x = p[x];
        }
        else
        {
            tmp = _query(1, 1, m, baseid[chead[vc]], baseid[y]);
            y = chead[vc];
```

```
            y = p[y];
        }
        ans = max(ans, tmp);
    }
    return ans;
}

int main()
{
    chead[0] = chead[n] = -1;
    dfs(1, 0);
    chain = 0;
    m = 0;
    hld(1, 0, 0);
    build(1, 1, m);
    update(a, b);
    query(a, b);
    return 0;
}
```

Centroid Decomposition

```
// Time Complexity - O(NlogN)
/* Centroid Decomposition of a tree */

/*-----------------Decomposition part--------------------------*/
int nn;
inline void dfs1(int v, int p)
{
    w[++nn] = v;
    sz[v] = 1;
    for (int i : vec[v])
    {
        if (!vis[i] && i != p)
        {
            dfs1(i, v);
            sz[v] += sz[i];
        }
    }
}

inline int dfs2(int v, int p)
{
    for (int u : vec[v])
    {
        if (!vis[u] && u != p && sz[u] > nn / 2)
        {
            return dfs2(u, v);
        }
    }
    return v;
}

inline void decompose(int v)
{
    nn = 0;
    dfs1(v, v);
    int centroid = dfs2(v, v);
    vis[centroid] = 1;

    for (int i = 0; i <= nn + 1; i++)
    {
        // reset only those node which are going to be checked in this subtree
    }

    for (int i : vec[centroid])
    {
        if (vis[i])
            continue;
        // call the function and perform the operation accordingly
    }

    for (int i = 0; i <= nn + 1; i++)
    {
        // after performing operation reset the array back to zero
    }

    for (int i : vec[centroid])
    {
        if (!vis[i])
            decompose(i);
    }
}
```