```
struct DSU{
	lld sze[N],arr[N];
	void init(){
		rep(i,1,N) arr[i]=i,sze[i]=1;
	}
	void get_union(lld a,lld b){
		lld root_a=root(a),root_b=root(b);
		if(sze[root_a]<sze[root_b])
	arr[root_a]=arr[root_b],sze[root_b]+=sze[root_a];
		else
	arr[root_b]=arr[root_a],sze[root_a]+=sze[root_b];
	}
	lld root(lld x){
		while(arr[x]!=x) arr[x]=arr[arr[x]],x=arr[x];
		return x;
	}
} dsu;
```

// Bipartite Matching - Hungarian Algorithm

// n - left, k - right

// assumes first n nodes in adj on the left

```
vector <vector <int> > adj;
class Kuhn{
public:
    int n, k;
    vector <vector<int> > g;
    vector <int> pairs_of_right, pairs_of_left;
    vector <bool> used;
    bool kuhn(int v){
        if(used[v]) return false;
        used[v] = true;
        for(int i = 0; i < g[v].size(); ++i){
            int to = g[v][i] - n;
            if(pairs_of_right[to] == -1 or
kuhn(pairs_of_right[to])){
                pairs_of_right[to] = v;
                pairs_of_left[v] = to;
                return true;
            }
        }
        return false;
    }
    vector <pair <int, int> > find_max_matching(vector
<vector <int> > &_g, int _n, int _k){
        g = _g;
        n = _n;
        k = _k;
        pairs_of_right = vector <int> (k, -1);
        pairs_of_left = vector <int> (n, -1);
        used = vector <bool> (n, false);
        bool path_found;
        do{
            fill(used.begin(), used.end(), false);
            path_found = false;
            for(int i = 0; i < n; ++i){
                if(pairs_of_left[i] < 0 and !used[i]){
                    path_found |= kuhn(i);
                }
            }
        } while(path_found);
        vector <pair <int, int> > res;
        for(int i = 0; i < k; ++i){
            if(pairs_of_right[i] != -1){
                res.pb(mp(pairs_of_right[i], i+n));
            }
        }
        return res;
    }
};
```

// PowerMod

```
template<typename T> T power(T x,T y,ll m=MOD){T
ans=1;while(y>0){if(y&1LL)
ans=(ans*x)%m;y>>=1LL;x=(x*x)%m;}return ans%m;}
```

// LCA

```
#define MAXN 2*100010
#define LOGMAXN 20
```

```
int T[MAXN]; // parent of node
int P[MAXN][LOGMAXN];
int L[MAXN]; // level of node
struct LCA{
	int n;
	void pre(){
		for(int i=0; i<n; i++){
			for(int j=0; (1<<j) < n; j++)
				P[i][j] = -1;
		}
		for(int i=0; i<n; i++) P[i][0] = T[i];
		for(int j=1; (1<<j)<n; j++){
			for(int i=0; i<n; i++){
				if(P[i][j-1] != -1){
					P[i][j] = P[P[i][j-1]][j-1];
				}
			}
		}
	}
	int query(int p, int q){
		int tmp, log;
		if(L[p] < L[q]) swap(p,q);
		for(log = 1; (1<<log) <= L[p]; log++);
		log--;
		for(int i=log; i>=0; i--){
			if(L[p] - (1<<i) >= L[q]){
				p = P[p][i];
			}
		}
		if(p==q) return p;
		for(int i=log; i>=0; i--){
			if(P[p][i]!=-1 and P[p][i]!=P[q][i]){
				p = P[p][i];
				q = P[q][i];
			}
		}
		return T[p];
	}
};
```

// Flow Dinic

```
template <typename T>
struct dinic{
    const T eps = (T)1e-9;
    struct edge{
        int to;
        T cap, flo;
        int rev;
    };
    vector <int> ptr, d;
    vector <vector <edge> > g;
    int n, source, sink;
    T flow;

    dinic(int n, int source, int sink) : n(n),
source(source), sink(sink){
        g.resize(n);
        ptr.resize(n);
        d.resize(n);
        flow = 0;
    }
    void clear(){
        flow = 0;
        for(int i = 0; i < n; ++i){
            for(auto& j: g[i]){
                j.flo = 0;
            }
        }
    }
    void AddEdge(int from, int to, T forward_capacity, T
backward_capacity = 0){
        //cout << from << ' ' << to << endl;
        int sz_to = g[to].size();
        int sz_frm = g[from].size();
        g[from].pb({to, forward_capacity, 0, sz_to});
        g[to].pb({from, backward_capacity, 0, sz_frm});
    }
    bool bfs(){
        queue <int> q;
```

```
        q.push(source);
        fill(d.begin(), d.end(), -1);
        d[source] = 0;
        while(!q.empty()){
            auto curr = q.front();
            q.pop();
            for(auto i: g[curr]){
                if(i.cap - i.flo > eps and d[i.to] == -1){
                    d[i.to] = d[curr] + 1;
                    if(i.to == sink) return true;
                    q.push(i.to);
                }
            }
        }
        return false;
    }
    T dfs(int v, T w){
        if(v == sink){
            return w;
        }
        while(ptr[v] >= 0){
            auto &e = g[v][ptr[v]];
            if(e.cap - e.flo > eps and d[e.to] == d[v]+1){
                T ret = dfs(e.to, min(e.cap - e.flo, w));
                if(ret > eps){
                    e.flo += ret;
                    g[e.to][e.rev].flo -= ret;
                    return ret;
                }
            }
            ptr[v]--;
        }
        return 0;
    }
    T GetMaxFlow(){
        while(bfs()){
            for(int i = 0; i < n; ++i){
                ptr[i] = g[i].size() - 1;
            }
            T inc = 0;
            while(1){
                T ret = dfs(source,
numeric_limits<T>::max());
                if(ret <= eps) break;
                inc += ret;
            }
            if(inc <= eps) break;
            flow += inc;
        }
        return flow;
    }
    vector <bool> getmincut(){
        GetMaxFlow();
        vector <bool> ret(n);
        for(int i = 0; i < n; ++i) ret[i] = (d[i] != -1);
        return ret;
    }
};
```

// Z FUNCTION

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

Number theoretic algorithms

```
// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
        int xx = y = 0;
        int yy = x = 1;
        while (b) {
```

```
        int q = a / b;
        int t = b; b = a%b; a = t;
        t = xx; xx = x - q*xx; x = t;
        t = yy; yy = y - q*yy; y = t;
    }
    return a;
}
// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
    int x, y;
    VI ret;
    int g = extended_euclid(a, n, x, y);
    if (!(b%g)) {
        x = mod(x*(b / g), n);
        for (int i = 0; i < g; i++)
            ret.push_back(mod(x + i*(n / g), n));
    }
    return ret;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
    int x, y;
    int g = extended_euclid(a, n, x, y);
    if (g > 1) return -1;
    return mod(x, n);
}

// Chinese remainder theorem (special case): find z such that
// z % m1 = r1, z % m2 = r2.  Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M).  On failure, M = -1.
PII chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
    int s, t;
    int g = extended_euclid(m1, m2, s, t);
    if (r1%g != r2%g) return make_pair(0, -1);
    return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
}

// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (m[i]).  Return (z, M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &m, const VI &r) {
    PII ret = make_pair(r[0], m[0]);
    for (int i = 1; i < m.size(); i++) {
        ret = chinese_remainder_theorem(ret.second, ret.first, m[i],
r[i]);
        if (ret.second == -1) break;
    }
    return ret;
}
// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
    if (!a && !b){
        if (c) return false;
        x = 0; y = 0;
        return true;
    }
    if (!a){
        if (c % b) return false;
        x = 0; y = c / b;
        return true;
```

```
	}
	if (!b){
		if (c % a) return false;
		x = c / a; y = 0;
		return true;
	}
	int g = gcd(a, b);
	if (c % g) return false;
	x = c / g * mod_inverse(a / g, b / g);
	y = (c - a*x) / b;
	return true;
}
```

// **MULTIPLICATIVE FUNCTIONS**

p, q coprime: f(pq) = f(p) * f(q)
f(p^k) = some repr (p = prime)
The Möbius function $\mu(p^k) = [k = 0] - [k = 1]$.
The Euler's totient function $\phi(p^k) = p^k - p^{k-1}$.

```
void eval_multiplicative_function(lld n){
	fill(is_composite, is_composite + n + 1, false);
	for(lld i = 2; i <= n; ++i){
		if(!is_composite[i]){
			prime.pb(i);
			func[i] = -1;
			cnt[i] = 1;
		}
		for(lld j = 0; j < prime.size() and i*prime[j] <= n; ++j){
			is_composite[i*prime[j]] = true;
			if(i%prime[j] == 0){
				func[i*prime[j]] = 0;
// func[i*p] = func[i/p^cnt[i]] * f(p^(cnt[i]+1))
				cnt[i*prime[j]] = cnt[i]+1;
				break;}
else{func[i*prime[j]] = func[i] * func[prime[j]]; cnt[i*prime[j]] = 1;}}}}
```

$$\sum_{d|n} \mu(d) = \epsilon(n) = [n = 1]$$

If $g(n) = \sum_{d|n} f(d)$ for every positive integer n, then $f(n) = \sum_{d|n} g(d)\mu(\frac{n}{d})$, where $\mu(x)$ is the Möbius function.

```
// GAUSSIAN ELIMINATION - BASIS
vector<int> gauss(vector<int> &v){
  vector<int>result;
  int base = 0;
  for(int i=30;i>=0;i--){
      int next = -1;
      for(int j=base;j<v.size();++j){
          if(v[j] & (1LL<<i)){
              next = j;
              break;
          }
      }
      if(next != -1){
          swap(v[base], v[next]);
          result.push_back(v[base]);
          base++;
          for(int j=base;j<v.size();++j){
              if(v[j]&(1<<i)) v[j] ^= v[base-1];
          }
      }
  }
  return result;
}
// MANACHER'S ALGORITHM
```

```
vector <int> manacher(string s){ // returns manacher's array the half length + center value
string t = "*";for(char i: s){t += i; t += '*';} int n = t.length();vector <int> p(n);int c = 0, r = -1, rad = 0; for(int i = 0; i < n; ++i){if(i <= r) rad = min(p[2*c - i], r - i) ;else rad = 1; while(i + rad < n and i - rad >= 0 and t[i - rad] == t[i + rad]) ++rad;p[i] = rad; if(i + rad - 1 > r) c = i, r = i + rad -1;}return p;}
```

Name	Original Recurrence	Sufficient	Origin	Optimi

		Condition of Applicability	**al Complexity**	**zed Complexity**	
Convex Hull Optimization1	$dp[i] = min_{j<i}\{dp[j] + b[j] \star a[i]\}$	$b[j] \geq b[j+1]$ $a[i] \leq a[i+1]$	$O(n^2)$	$O(n)$	
Convex Hull Optimization2	$dp[i][j] = min_{k<j}\{dp[i-1][k] + b[k] * a[j]\}$	$b[k] \geq b[k+1]$ $a[j] \leq a[j+1]$	$O(kn^2)$	$O(kn)$	
Divide and Conquer Optimization	$dp[i][j] = min_{k<j}\{dp[i-1][k] + C[k][j]\}$	$A[i][j] \leq A[i][j+1]$	$O(kn^2)$	$O(knlogn)$	
Knuth Optimization	$dp[i][j] = min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$	$A[i, j-1] \leq A[i, j] \leq A[i+1, j]$	$O(n^3)$	$O(n^2)$	

$A[i][j]$ — 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 Optimization2** 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**: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c],\ a \leq b \leq c \leq d$
- **monotonicity**: $C[b][c] \leq C[a][d],\ a \leq b \leq c \leq d$
- It is claimed (in the references) that the recurrence $dp[j] = min_{i<j}\{dp[i] + C[i][j]\}$ can be solved in $O(nlogn)$ (and even $O(n)$) if $C[i][j]$ satisfies **quadrangle inequality**

// EXAMPLES: KNUTH'S

```
// conditions: C[a][d] >= C[b][c] (a < b < c < d)
// and C[a][c] + C[b][d] <= C[a][d] + C[b][c]
// for dp[i][j] = min(k) (dp[i][k] + dp[k][j]) + C[i][j]
for(int i = n; i >= 0; --i){
     for(int j = i; j <= n; ++j){
          if(j <= i+1){
               dp[i][j] = 0;
               md[i][j] = i;
          }
          else{
               int mleft = md[i][j-1];
               int mright = md[i+1][j];
               dp[i][j] = inf;
          for(int r = mleft; r <= mright; ++r){
               lld tmp = dp[i][r] + dp[r][j] + sm[j] -
sm[i];
               if(tmp < dp[i][j]){
                    dp[i][j] = tmp;
                    md[i][j] = r;
```

```
}}}}}.
```

// Divide and Conquer Optimization

```
// dp[j][i] = min(k<j) (dp[k][i-1] + C[k+1][j])
// A[j][i] := optimal k for dp[j][i]
// for all i, j: A[j][i] <= A[j+1][i]
// O(m*n*n) => O(m*n*logn)
// initialise dp for i = 1 somehow
// for i in [2:n] -> calc(i, 1, n, 1, n)
// dp: calculated layer by layer (layers of i)
void calc(int i, int lo, int hi, int optl, int optr){
      if(lo > hi) return;
      int md = (lo + hi)/2;
      int bestk = -1;
      dp[md][i] = inf;
      for(lld r = optl; r <= optr;++r){
            lld tmp = dp[r][i-1] + C[r+1][md];
            if(tmp < dp[md][i]){
                  dp[md][i] = tmp;
                  bestk = r;
            }
      }
      calc(i, lo, md-1, optl, bestk);
      calc(i, md+1, hi, bestk, optr);
}
```

//D&C PseudoCode

```
def ComputeDP(i, jleft, jright, kleft, kright):
# Select the middle point
jmid = (jleft + jright) / 2
# Compute the value of dp[i][jmid] by definition of DP
dp[i][jmid] = +INFINITY
bestk = -1
for k in range(kleft, jmid):
  if dp[i - 1][k] + C[k + 1][jmid] < best:
    dp[i][jmid] = dp[i - 1][k] + C[k + 1][jmid]
        bestk = k
    # Divide and conquer
    if jleft < jmid - 1:
      ComputeDP(i, jleft, jmid - 1, kleft, bestk)
    if jleft + 1 < jright:
      ComputeDP(i, jmid + 1, jright, bestk, kright)

    def ComputeFullDP:
    Initialize dp for i = 0 somehow
    for i in range(1, m):
        ComputeDP(i, 0, n, 0, n)
```

// CONVEX HULL OPTIMISATION

```
vector<pi>L; // stack of lines
bool bad(pi a,pi b,pi c) {
    return (double)(a.s-b.s)*(double) (c.f-a.f) <(double)
(a.s-c.s)*(double)(b.f-a.f)  ;
}
void add(ll m,ll c) {
    int sz;
    while(L.size()>=2) {
        sz=L.size()-1;
        if(bad( L[sz],L[sz-1],mp(m,c) )  ) {
            L.pop_back();
        } else break;
    }
    L.pb(mp(m,c));
}
int pt=0;
ll query(ll x) {
    pt=min(pt,(int)L.size()-1);
    while(pt+1<L.size() and L[pt+1].f*x+L[pt+1].s <
L[pt].f*x+L[pt].s ) pt++;
    return L[pt].f * x + L[pt].s;
}
```

```
// BRIDGE TREE
queue <int> Q[N];
vector <int> tree[N], graph[N]; // edge list representation
int U[M], V[M];
int tim[N]; // stores time stamp
int stamp;
bool isbridge[M];
bool vis[N];

int getvertex(int u, int e){
	return (U[e] == u ? V[e] : U[e]);
}

int predfs(int u, int e){ // identify bridges
	vis[u] = 1;
	tim[u] = stamp++;
	int mxs = tim[u];
	for(auto i: graph[u]){
		if(i == e) continue;
		int w = getvertex(u, i);
		if(!vis[w]) mxs = min(mxs, predfs(w, i));
		else mxs = min(mxs, tim[w]);
	}
	if(mxs == tim[u] and e != -1) isbridge[e] = 1;
	return mxs;
}

void dfs(int u){ // construct bridge tree
	int currcmp = cmpno; // current component number
	Q[currcmp].push(u);
	vis[u] = 1;
	while(!Q[currcmp].empty()){
		int v = Q[currcmp].front();
		Q[currcmp].pop();
		for(auto i: graph[v]){
			int w = getvertex(v, i);
			if(vis[w]) continue;
			if(isbridge[i]){
				cmpno++;
				tree[currcmp].pb(cmpno);
				tree[cmpno].pb(currcmp);
				dfs(w);
			}
			else{
				Q[currcmp].push(w);
				vis[w] = 1;
			}
		}
	}
}
```

------------------------[GEOMETRY MISC]---------------------------

```
ldb inf = 1e100;
ldb eps = 1e-12;

struct PT{
    ldb x, y;
    PT() {}
    PT(ldb x, ldb 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 * (ldb c) const { return PT(x*c, y*c ); }
    PT operator / (ldb c) const { return PT(x/c, y/c ); }
     bool operator<(const PT &rhs) const { return mp(y,x) < mp(rhs.y,rhs.x); }
     bool operator==(const PT &rhs) const { return mp(y,x) == mp(rhs.y,rhs.x); }
};

ldb dot(PT p, PT q){ return p.x*q.x + p.y*q.y; }
ldb cross(PT p, PT q) { return p.x*q.y - p.y*q.x; }
ldb normsq(PT p){ return dot(p, p); }
ldb dist2(PT p, PT q) { return normsq(p-q); }
```

```
ostream &operator << (ostream &os, const PT &p){
    os << "(" << p.x << "," << p.y << ")";
}

PT RotateCCW90(PT p) { return PT(-p.y, p.x); }
PT RotateCW90(PT p) { return PT(p.y, -p.x); }
PT RotateCCW(PT p, ldb t){
    return PT(p.x*cos(t) - p.y*sin(t), p.x*sin(t) +
p.y*cos(t));
}

PT ProjectPointLine(PT a, PT b, PT c){
    // project c on line through a and b
    // assert(a!=b);
    return a + (b-a)*dot(b-a, c-a)/normsq(b-a);
}

PT ProjectPointSegment(PT a, PT b, PT c){
    // return point closest to c on segment a --- b
    ldb r = normsq(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;
}

ldb DistancePointSegment(PT a, PT b, PT c){
    // distance of point c from segment a --- b
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

bool LinesParallel(PT a, PT b, PT c, PT d){
    return fabs(cross(a-b, 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;
}

bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if(LinesCollinear(a, b, c, d)) {
        if(dist2(a, c) < eps or dist2(a, d) < eps or
dist2(b, c) < eps or dist2(b, d) < eps) return true;
        if(dot(c-a, c-b) > 0 and dot(d-a, d-b) > 0 and
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 and 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 and q.y < p[j].y or p[j].y <= q.y
and q.y < p[i].y) and 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, ldb r) {
    vector<PT> ret;
    b = b-a;
    a = a-c;
    ldb A = dot(b, b);
    ldb B = dot(a, b);
    ldb C = dot(a, a) - r*r;
    ldb D = B*B - A*C;
    if (D < -eps) return ret;
    ret.pb(c+a+b*(-B+sqrt(D+eps))/A);
    if (D > eps)
    ret.pb(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, ldb r, ldb
R) {
    vector<PT> ret;
    ldb d = sqrt(dist2(a, b));
    if (d > r+R or d+min(r, R) < max(r, R)) return ret;
    ldb x = (d*d-R*R+r*r)/(2*d);
    ldb y = sqrt(r*r-x*x);
    PT v = (b-a)/d;
    ret.pb(a+v*x + RotateCCW90(v)*y);
    if(y > 0) ret.pb(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".
ldb ComputeSignedArea(const vector<PT> &p) {
    ldb 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;
}
ldb ComputeArea(const vector<PT> &p) {
    return fabs(ComputeSignedArea(p));
}
PT ComputeCentroid(const vector<PT> &p) {
```

```
    PT c(0,0);
    ldb 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 or j == k) continue;
            if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
            return false;
        }
    }
    return true;
}

// compute distance between point (x,y,z) and plane
ax+by+cz=d
ldb DistancePointPlane(ldb x, ldb y, ldb z, ldb a, ldb b,
ldb c, ldb d) {
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}
ldb area2(PT a, PT b, PT c){
     return cross(a,b) + cross(b,c) + cross(c,a);
}
```

```
// CONVEX HULL
// area2 and cross functions required
// INPUT: vector of points (unordered)
// OUTPUT:      a vector of points of convex hull,
counterclockwise, starting with bottommost/leftmost point
vector <PT> Hull(vector <PT> pts){
     sort(pts.begin(),pts.end());
     vector <PT> up, dn, H;
     int sz = pts.size();
     for(int i=0; i<sz; i++){
          while(up.size() > 1 && area2(up[up.size()-
2],up.back(),pts[i])>=0) up.pop_back();
          while(dn.size() > 1 && area2(dn[dn.size()-
2],dn.back(),pts[i])<=0) dn.pop_back();
          up.pb(pts[i]);
          dn.pb(pts[i]);
     }
     H = dn;
     sz = up.size();
     for(int i = sz - 2; i>=1; i--) H.pb(up[i]);
     return H;
}
```

```
// KOSARAJU
vector <pair <int, int> > edges;
stack <int> stk;
vector <int> adj[N], adj2[N];
bool vis[N];
int component[N];

void dfs(int i){
    vis[i] = 1;
    for(auto j: adj[i]){
        if(!vis[j]) dfs(j);
    }
    stk.push(i);
}

void dfs2(int i, int stp){
    vis[i] = 1;
    for(auto x: adj2[i]){ // reversed edges
        if(vis[x]) continue;
        dfs2(x, stp);
    }
    component[i] = stp;
}
```

```
main(){
    int n, m;
    cin >> n >> m;
    for(int i = 0, x, y; i < m; ++i){
        cin >> x >> y;
        adj[x].pb(y);
        adj2[y].pb(x);
    }
    for(int i = 1; i <= n; ++i){
        if(!vis[i]) dfs(i);
    }
    for(int i = 1; i <= n; ++i){
        vis[i] = 0;
    }
    int cntr = 0;
    while(!stk.empty()){
        auto t = stk.top();
        stk.pop();
        if(vis[t]) continue;
        cntr++;
        dfs2(t, cntr);
    }
    for(int i = 1; i <= n; ++i){
        trace(i, component[i]);
    }
}
```

```
// MINCOST MAXFLOW
struct MCMF {
  typedef lld ctype;
  struct Edge { lld x, y; lld cap, cost; };
  vector<Edge> E;      vector<lld> adj[MAXN];
  lld N, prev[MAXN];   ctype dist[MAXN], phi[MAXN];

  MCMF(lld NN) : N(NN) {}

  void add(lld x,lld y,ctype cap,ctype cost) {  // cost >= 0
    cost += EPS;
      //   printf("Adding (%d, %d) having (%d, %d)\n", x, y,
cap, cost);
    Edge e1={x,y,cap,cost}, e2={y,x,0,-cost};
    adj[e1.x].push_back(E.size()); E.push_back(e1);
    adj[e2.x].push_back(E.size()); E.push_back(e2);
  }

  void mcmf(lld s,lld t, ctype &flowVal, ctype &flowCost) {
    lld x;
    flowVal = flowCost = 0;  memset(phi, 0, sizeof(phi));
    while (true) {
      for (x = 0; x < N; x++) prev[x] = -1;
      for (x = 0; x < N; x++) dist[x] = INF;
      dist[s] = prev[s] = 0;

      set< pair<ctype, lld> > Q;
      Q.insert(make_pair(dist[s], s));
      while (!Q.empty()) {
        x = Q.begin()->second; Q.erase(Q.begin());
        tr(it,adj[x]) {
          const Edge &e = E[*it];
          if (e.cap <= 0) continue;
          ctype cc = e.cost + phi[x] - phi[e.y];
// ***
          if (dist[x] + cc + EPS < dist[e.y]) {
            Q.erase(make_pair(dist[e.y], e.y));
            dist[e.y] = dist[x] + cc;
            prev[e.y] = *it;
            Q.insert(make_pair(dist[e.y], e.y));
          }
        }
      }
      if (prev[t] == -1) break;

      ctype z = INF;
      for (x = t; x != s; x = E[prev[x]].x) z = min(z,
E[prev[x]].cap);
      for (x = t; x != s; x = E[prev[x]].x)
        { E[prev[x]].cap -= z; E[prev[x]^1].cap += z; }
```

```
      flowVal += z;
      flowCost += z * (dist[t] - phi[s] + phi[t]);
      for (x = 0; x < N; x++) if (prev[x] != -1) phi[x] +=
dist[x];     // ***
    }
  }
};
```

// FAST FOURIER TRANSFORM

```
typedef complex<double> base;
const double PI = 4*atan(1);
struct FFT {
  vector<base> omega;
  long long FFT_N;
  void init_fft(long long n) {
    FFT_N  = n;
    omega.resize(n);
    double angle = 2 * PI / n;
    for(int i = 0; i < n; i++)
      omega[i] = base( cos(i * angle), sin(i * angle));
  }
  void fft (vector<base> & a) {
    long long n = (long long) a.size();
    if (n == 1)  return;
    long long 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);
    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;
    }
  }
  void multiply (const vector<long long> & a, const vector<long long> & b,
vector<long long> & 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);
    fft (fa),  fft (fb);
    for (size_t i=0; i<n; ++i)
      fa[i] = conj( fa[i] * fb[i]);
    fft (fa);
    res.resize (n);
    for (size_t i=0; i<n; ++i) {
      res[i] = (long long) (fa[i].real() / n + 0.5);
      res[i]%=mod;
    }
  }
};
```

// KMP

```
void process(string str){
      int n = str.length();
      pre[0] = 0;
      for(int j=0, i=1; i<n; i++){
            while(j>0 and str[i]!=str[j]) j = pre[j-1];
            if(str[i]==str[j]) j++;
            pre[i] = j;
      }
}
int kmp(string s){
      process(s);
      int i = 0, j = 0,n = text.length(),m = s.length();
      while(1){
            if(j==n) return -1;
            if(text[j]==s[i]){
                  i++; j++;
                  if(i==m) return j-i;
            }
```

```
        else if(i>0) i = pre[i];
else j++;
}
}
```

//Fully Dynamic Convex Hull Trick

/* Given a set of pairs (m, b) specifying lines of the form y = mx + b, process a set of x-coordinate queries each asking to find the minimum y-value when any of the given lines are evaluated at the specified x. To instead have the queries optimize for maximum y-value, call the constructor with query_max=true.
The following implementation is a fully dynamic variant of the convex hull optimization technique, using a self-balancing binary search tree (std::set) to support the ability to call add_line() and query() in any desired order.
Time Complexity:
- O(n) for any interlaced sequence of add_line() and query() calls, where n is the number of lines added. This is because the overall number of steps taken by add_line() and query() are respectively bounded by the number of lines. Thus a single call to either add_line() or query() will have an O(1) amortized running time.
Space Complexity:
- O(n) for storage of the lines.
- O(1) auxiliary for add_line() and query().
*/

```
#include <limits>
#include <set>
class hull_optimizer {
 struct line {
   long long m, b, value;
   double xlo;
   bool is_query, query_max;

   line(long long m, long long b, long long v, bool
is_query, bool query_max)
       : m(m), b(b), value(v), xlo(-
std::numeric_limits<double>::max()),
         is_query(is_query), query_max(query_max) {}

   double intersect(const line &l) const {
     if (m == l.m) {
       return std::numeric_limits<double>::max();
     }
     return (double)(l.b - b)/(m - l.m);
   }

   bool operator<(const line &l) const {
     if (l.is_query) {
       return query_max ? (xlo < l.value) : (l.value < xlo);
     }
     return m < l.m;
   }
 };

 std::set<line> hull;
 bool query_max;

 typedef std::set<line>::iterator hulliter;

 bool has_prev(hulliter it) const {
   return it != hull.begin();
 }

 bool has_next(hulliter it) const {
   return (it != hull.end()) && (++it != hull.end());
 }
bool irrelevant(hulliter it) const {
   if (!has_prev(it) || !has_next(it)) {
     return false;
   }
   hulliter prev = it, next = it;
   --prev;
   ++next;
   return query_max ? (prev->intersect(*next) <= prev-
>intersect(*it))
                    : (next->intersect(*prev) <= next-
>intersect(*it));
 }

 hulliter update_left_border(hulliter it) {
```

```
   if ((query_max && !has_prev(it)) || (!query_max &&
!has_next(it))) {
     return it;
   }
   hulliter it2 = it;
   double value = it->intersect(query_max ? *--it2 :
*++it2);
   line l(*it);
   l.xlo = value;
   hull.erase(it++);
   return hull.insert(it, l);
 }

public:
 hull_optimizer(bool query_max = false) :
query_max(query_max) {}

 void add_line(long long m, long long b) {
   line l(m, b, 0, false, query_max);
   hulliter it = hull.lower_bound(l);
   if (it != hull.end() && it->m == l.m) {
     if ((query_max && it->b < b) || (!query_max && b < it-
>b)) {
       hull.erase(it++);
     } else {
       return;
     }
   }
   it = hull.insert(it, l);
   if (irrelevant(it)) {
     hull.erase(it);
     return;
   }
   while (has_prev(it) && irrelevant(--it)) {
     hull.erase(it++);
   }
   while (has_next(it) && irrelevant(++it)) {
     hull.erase(it--);
   }
   it = update_left_border(it);
   if (has_prev(it)) {
     update_left_border(--it);
   }
   if (has_next(++it)) {
     update_left_border(++it);
   }
 }

 long long query(long long x) const {
   line q(0, 0, x, true, query_max);
   hulliter it = hull.lower_bound(q);
   if (query_max) {
     --it;
   }
   return it->m*x + it->b;
 }
};
/*** Example Usage ***/
#include <cassert>
int main() {
 hull_optimizer h;
 h.add_line(3, 0);
 h.add_line(0, 6);
 h.add_line(1, 2);
 h.add_line(2, 1);
 assert(h.query(0) == 0);
 assert(h.query(2) == 4);
 assert(h.query(1) == 3);
 assert(h.query(3) == 5);
 return 0;
}
```

// TREAP

```
struct node{
   int val, prior, size;
   node *l, *r;
};
typedef node* pnode;
```

```
pnode getnew(int x){
   int y = rand();
   pnode ret = new node;
   ret->prior = y;
   ret->val = x;
   ret->size = 1;
   ret->l = ret->r = NULL;
   return ret;
}
int sz(pnode x){
   return x ? x->size : 0;
}
void upd_sz(pnode t){
   if(t){
       t->size = sz(t->l) + 1 + sz(t->r);
   }
}
void split(pnode t, pnode& l, pnode& r, int key){
   if(!t) l = r = NULL;
   else if(t->val <= key){
       split(t->r, t->r, r, key);
       l = t;
   }
   else{
       split(t->l, l, t->l, key);
       r = t;
   }
   upd_sz(t);
}
void merge(pnode& t, pnode l, pnode r){
   if(!l or !r) t = l ? l : r;
   else if(l->prior > r->prior){
       merge(l->r, l->r, r);
       t = l;
   }
   else{
       merge(r->l, l, r->l);
       t = r;
   }
   upd_sz(t);
}
void insert(pnode& t, pnode x){
   if(!t) t = x;
   else if(t->prior > x-> prior){
       if(t->val <= x->val){
           insert(t->r, x);
       }
       else{
           insert(t->l, x);
       }
   }
   else{
       split(t, x->l, x->r, x->val);
       t = x;
   }
   upd_sz(t);
}
void erase(pnode& t, int x){
   if(!t) return;
   if(t->val == x){
       pnode tmp = t;
       merge(t, t->l, t->r);
       free(tmp);
   }
   else{
       if(t->val < x) erase(t->r, x);
       else erase(t->l, x);
   }
   upd_sz(t);
}
main(){
   int n, m;
   cin >> n >> m;
}
```

//Implicit Treap

```
struct node{
     int size, prior;
```

```
	int lazy; // for lazy updates
	int sum; // ans to query as per usage
	int val; // value stored in the array
	node *l, *r;
};

typedef node* pnode;

pnode getnew(int x){
	pnode ret = new node;
	ret->prior = rand();
	ret->val = x;
	ret->lazy = 0;
	ret->size = 1;
	ret->l = ret->r = NULL;
	return ret;
}

int sz(pnode t){
	return t ? t->size : 0;
}
void upd_sz(pnode t){
	if(t) t->size = sz(t->l) + 1 + sz(t->r);
}


// lazy propagation
void lazy(pnode t){
	if(!t or !t->lazy) return;
	t->val += t->lazy; // operation of lazy
	t->sum += t->lazy*sz(t);
	if(t->l) t->l->lazy += t->lazy;
	if(t->r) t->r->lazy += t->lazy;
	t->lazy = 0;
}

void reset(pnode t){
	if(t) t->sum = t->val;
}

// calculate answer while combining nodes, here sum is
returned
void combine(pnode&t, pnode l, pnode r){
	if(!l or !r) t = l ? l : r;
	t->sum = l->sum + r->sum; //  can be replaced with any
other operation
}

void operation(pnode t){
	if(!t) return;
	reset(t);
	lazy(t->l); lazy(t->r);
	combine(t, t->l, t);
	combine(t, t, t->r);
}

void split(pnode t, pnode& l, pnode& r, int pos, int add =
0){
	if(!t) l = r = NULL;
	else{
		lazy(t);
		int cur_pos = add + sz(t->l);
		if(cur_pos <= pos){
			split(t->r, t->r, r, pos, cur_pos + 1);
			l = t;
		}
		else{
			split(t->l, l, t->l, pos, add);
			r = t;
		}
		upd_sz(t);
		operation(t);
	}
}
void merge(pnode& t, pnode l, pnode r){
lazy(l);lazy(r);if(!l or !r) t = l ? l : r;
else if(l->prior > r->prior) merge(l->r, l->r, r), t = l;
else merge(r->l, l, r->l), t =r;upd_sz(t);operation(t);}
```

```
int range_query(pnode t, int l, int r){ // [l,r]
pnode L, mid, R; split(t, L, mid, l-1); split(mid, t, R, r-
l); int ans = t->sum; merge(mid, L, t);merge(t, mid, R);
return ans;}
void range_update(pnode t, int l, int r, int val){ // [l,r]
pnode L, mid, R;split(t, L, mid, l-1);split(mid, t, R, r-l);
t->lazy += val;merge(mid, L, t);merge(t, mid, R);}
```

// HLD

```
void dfs(int u, int p = 0) {
   size[u] = 1;
   parent[u] = p;
   for (auto v : g[u]) if (v != p) {
       depth[v] = depth[u] + 1;
       dfs(v, u);
       size[u] += size[v];
       if (!hld_child[u] || size[hld_child[u]] < size[v])
          hld_child[u] = v;
   }
}
// gives a 1-index to each node such that indices
// in each heavy path are contiguous
void hld(int u, int p = 0) {
   static int index = 0;
   hld_index[u] = ++index;
   hld_order[hld_index[u]] = X[u];
   if (!hld_root[u])
       hld_root[u] = u;
   if (hld_child[u]) {
       hld_root[hld_child[u]] = hld_root[u];
       hld(hld_child[u], u);
   }
   for (auto v : g[u])
       if (v != p && v != hld_child[u])
           hld(v, u);
}
// perform a query the path betwwen a and b,
// where query_path is a function on ranges of hld indices
void hld_query(int a, int b) {
   int a_value = 0, b_value = 0;
   while (hld_root[a] != hld_root[b]) {
       if (depth[hld_root[a]] < depth[hld_root[b]]) {
           b_value += query_path(hld_index[hld_root[b]],
               hld_index[b]);
           b = parent[hld_root[b]];
       }
       else {
           a_value +=
query_path(hld_index[hld_root[a]],hld_index[a]);
           a = parent[hld_root[a]];
       }
   }
   if (depth[a] < depth[b])
       b_value += query_path(hld_index[a], hld_index[b]);
   else
       a_value += query_path(hld_index[b], hld_index[a]);
   return a_value + b_value;
}
```

// CENTROID DECOMPOSITION

```
void dfs1(lld curr,lld par) {
   child[curr]=1,total++;
   for(auto i:adj[curr])
       if(i!=par) {
           dfs1(i,curr);
           child[curr]+=child[i];}}
lld dfs2(lld curr,lld par) {
   for(auto i:adj[curr])
       if(i!=par and
           child[i]>(total/2))
           return dfs2(i,curr);
   return curr;
}
void decompose(lld curr,lld par) {
   total=0,dfs1(curr,curr);
   lld centroid=dfs2(curr,curr);
   if(par==0) par=centroid,root=centroid;
   parent[centroid]=par;
```

```
    for(auto i:adj[centroid]) {
        adj[i].erase(centroid);
        decompose(i,centroid);
    }
    adj[centroid].clear();
}
void update(lld curr) {
    lld tmp=curr;
    while(1)
    {
        ans[tmp]=min(ans[tmp],
            lc.dist(curr,tmKMPp));
        tmp=parent[tmp];
        if(tmp==root) {
            ans[tmp]=min(ans[tmp],
                lc.dist(curr,tmp));
            break;
        }
    }
}
```

```
// PARALLEL BINARY SEARCH
lld bound=log2(k);
 rep(i,0,bound+1) {
    ft.init();
    rep(i,1,n+1) if(low[i]!=high[i])
check[(low[i]+high[i])/2].pb(i);
    rep(i,1,k+1) {
      if(l[i]<=r[i]) ft.update(l[i],a[i]),ft.update(r[i]+1,-
a[i]);
      else ft.update(1,a[i]),ft.update(r[i]+1,-
a[i]),ft.update(l[i],a[i]);
      while(sz(check[i])) {
        lld curr=check[i].back();
        check[i].pop_back();
        lld curr_sum=0;
        for(auto j:par[curr]) {
          curr_sum+=ft.query(j);
          if(curr_sum>=p[curr]) break;
        }
        if(curr_sum>=p[curr]) high[curr]=i;
        else low[curr]=i+1;}}}
```

```
// LAZY
vector <lld> tree;
vector <lld> lazy;
void update(lld node, lld start, lld end, lld val, lld l,
lld r){
    if(lazy[node]){
        tree[node] += (end - start + 1)*lazy[node];
        if(start != end){
            lazy[node<<1] += lazy[node];
            lazy[node<<1 | 1] += lazy[node];
        }
        lazy[node] = 0;
    }
    if(start > r || end < l || start > end) return;
    if(start >= l && end <= r){
        tree[node] += (end - start + 1)*val;
        if(start != end){
            lazy[node<<1] += val;
            lazy[node<<1 | 1] += val;
        }
        return;
    }
    lld mid = (start + end) / 2;
    update(node<<1, start, mid, val, l, r);
    update(node<<1 | 1, mid+1, end, val, l, r);
    tree[node] = tree[node<<1] + tree[node << 1 | 1];
}
lld query(lld node, lld start, lld end, lld l, lld r){
    if(start > r || end < l || start > end) return 0;
    if(lazy[node]){
        tree[node] += (end - start + 1)*lazy[node];
        if(start!=end){
            lazy[node<<1] += lazy[node];
            lazy[node<<1 | 1] += lazy[node];
        }
        lazy[node] = 0;
```

```
    }
    if(start >= l && end <= r){
        return tree[node];
    }
    lld mid = (start + end) / 2;
    lld p1 = query(node<<1 , start, mid, l, r);
    lld p2 = query(node<<1 | 1, mid+1, end, l, r);
    return p1 + p2;
}
```

//Hopcroft Karp

```
int N, matched[2 * MAXN], dist[2 * MAXN], pt[MAXN];
vector<int> g[MAXN];
bool bfs() {
fill(dist, dist + 2 * N, -1);
queue<int> q;
REP (i, N) if (!matched[i]) {
dist[i] = 0;
q.push(i);
}
bool found = false;
while (!q.empty()) {
int u = q.front(); q.pop();
if (u > N && !matched[u]) found = true;
if (u <= N) { // left side
for (auto v : g[u])
if (dist[v] == -1) {
dist[v] = dist[u] + 1;
q.push(v); }
} else if (u > N && matched[u]) { // right side
if (dist[matched[u]] == -1) {
dist[matched[u]] = dist[u] + 1;
q.push(matched[u]);
} } }
return found;
}
bool dfs(int u) {
for (int &i = pt[u]; i < g[u].size(); ++i) {
int v = g[u][i];
if (dist[v] == dist[u] + 1) {
if (!matched[v] || (dist[matched[v]] == dist[v] + 1 &&
dfs(matched[v]))) {
matched[v] = u;
matched[u] = v;
return true;
} } }
return false;
}
int hopcroft_karp() {
int total = 0;
while (bfs()) {
fill(pt, pt + N, 0);
REP (i, N)
if (!matched[i])
if (dfs(i)) ++total;
}
return total;
}
```

// ORDERED STATISTICS TREE

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
typedef
tree<int,null_type,less<int>,rb_tree_tag,tree_order_statisti
cs_node_update> order_set;
order_set x;
int32_t main(){
      x.insert(5);
      x.insert(7);
      cout<<x.order_of_key(5)<<endl;
      cout<<x.order_of_key(8)<<endl; // strictly lesser
      cout<<*x.find_by_order(0)<<endl; // kth index, starts
from zero index, acsending
}
```

//2D Compressed BIT

```
order_set bit[N];
void insert(int x,int y){
```

```
	for(int i=x;i<N;i+=i&-i)
		bit[i].insert(mp(y,x));
}
void erase(int x,int y){
	for(int i=x;i<N;i+=i&-i)
		bit[i].erase(mp(y,x));
}
int get(int x,int y){
	int ans=0;
	for(int i=x;i>0;i-=i&-i)
		ans+=bit[i].order_of_key(mp(y+1,0));
	return ans;
}
```

//Sum of GP in LogN

```
ll solve(ll x,ll n,ll m){
//	trace3(x,n,m);
	if(n==0) return 1LL;
	if(n==1) return (1LL+x)%m;
	if(n%2==0){
		ll t1=solve((x*x)%m,n/2LL-1LL,m);
		t1=(t1*(1LL+x))%m;
		t1=(t1+power(x,n,m))%m;
		return t1;
	}
	else{
		ll t1=solve((x*x)%m,n/2LL,m);
		t1=(t1*(1LL+x))%m;
		return t1;
	}
}
```

```
// PALINDROMIC TREE
struct node{
	int next[26];
	int len;
	int sufflink;
	int ct=0;
};
node tree[N/10];
string s;
int num; //1-odd root,len -1,,,,,, 2-even root, len 0.
int prevnode;
void initree(){
	num=2; prevnode=2;
	tree[1].len=-1; tree[2].len=0;
	tree[1].sufflink=1; tree[2].sufflink=1;
}
void add(int i){
	int cur=prevnode,curlen=0,val=s[i]-'a';
	while(1){
		curlen=tree[cur].len;
		if(i-1-curlen>=0 && s[i-1-curlen]==s[i]) break;
		cur=tree[cur].sufflink;
	}
	if(tree[cur].next[val]){
		prevnode=tree[cur].next[val];
		return;
	}
	++num;
	prevnode=num;
	tree[num].len=tree[cur].len+2;
	tree[cur].next[val]=num;
	if(tree[num].len==1){
		tree[num].sufflink=2;
		tree[num].ct=1;
		return;
	}
	while(1){
		cur=tree[cur].sufflink;
		curlen=tree[cur].len;
		if(i-1-curlen>=0 && s[i-1-curlen]==s[i]){
			tree[num].sufflink=tree[cur].next[val];
			break;
		}
	}
	tree[num].ct=1+tree[tree[num].sufflink].ct;
}
int32_t main(){
```

```
	initree();
	cin>>s;
	int ans=0;
	int siz=s.size();
	rep(i,0,siz){
		add(i);
		ans+=tree[prevnode].ct;
		cout<<num-2<<" ";
	}
}
```

// TRIE

```
const int cn=2;
const int lastbit=30;
int NEW;
typedef struct node{
	int edges[cn];
}Trie;
Trie trie[N];
void initialize(int ind){
	rep(i,0,cn) {
		trie[ind].edges[i]=-1;
	}
}
void pretrie(){
	initialize(0); NEW++;
}
int ch(int x,int i){
	if(x&(1LL<<i)) return true;
	return false;
}
void insert(int x){
	int ind=0,curr;
	for(int i=lastbit;i>=0;i--){
		curr=ch(x,i);
		if(trie[ind].edges[curr]==-1){
			initialize(NEW);
			trie[ind].edges[curr]=NEW++;
		}
		ind=trie[ind].edges[curr];
	}
}
int fmax(int x){
	int ind=0,curr,req;
	for(int i=lastbit;i>=0;i--){
		curr=ch(x,i);
		req=curr^1LL;
		if(trie[ind].edges[req]!=-1){
			x|=(1LL<<i);
			ind=trie[ind].edges[req];
		}
		else{
			x&=~(1LL<<i);
			ind=trie[ind].edges[curr];
		}
	}
	return x;
}
int32_t main(){int l,r; cin>>l>>r;int maxi=0;pretrie();
rep(i,l,r+1) insert(i);rep(i,l,r+1){ maxi=max(maxi,fmax(i));
}cout<<maxi<<endl;}
```

// MO'S ALGORITHM

```
lld k,pref[N],ans[N],a[N],curr_ans,cnt[1<<20];
lld BLOCK_SIZE;
pair<pair<long long,long long>,long long> queries[100005];
bool my_comp(pair<pair<long long,long long>,long long>
x,pair<pair<long long,long long>,long long> y)
{
long long block_x=x.first.first/BLOCK_SIZE;
long long block_y=y.first.first/BLOCK_SIZE;
if(block_x!=block_y)
return block_x<block_y;
return x.first.second<y.first.second;
}
```

```
inline void add(long long x) {//Code for Add}
inline void remove(long long x) {//Code for Remove}

int main() {
long long n,m,i,j;
cin>>n>>m>>k;
pref[0]=0;
rep(i,1,n+1) cin>>a[i],pref[i]=pref[i-1]^a[i];
BLOCK_SIZE=static_cast<long long>(sqrt(n));
rep(i,0,m) {
cin>>queries[i].first.first>>queries[i].first.second;
queries[i].second=i;
queries[i].first.first--;
}
sort(queries,queries+m,my_comp);
long long left,right,currl=0,currr=-1;
rep(i,0,m) {
left=queries[i].first.first;
right=queries[i].first.second;
while(currr<right) add(pref[++currr]);
while(currr>right) remove(pref[currr--]);
while(currl<left) remove(pref[currl++]);
while(currl>left) add(pref[--currl]);
ans[queries[i].second]=curr_ans;
}rep(i,0,m) cout<<ans[i]<<endl;return 0;}
```

// PERSISTENT SEG TREE

```
struct node {int count;node *left, *right;
node(int count, node *left, node *right):
count(count), left(left), right(right) {}
node* insert(int l, int r, int w);};
node *null = new node(0, NULL, NULL);
node * node::insert(int l, int r, int w) {
if(l <= w && w < r) {if(l+1 == r) {return new node(this-
>count+1, null, null);}int m = (l+r)>>1;
return new node(this->count+1, this->left->insert(l, m, w),
this->right->insert(m, r, w));}return this;}node *root[N];
null->left = null->right = null;
```

// MATRIX EXPO

```
struct matrix {
int n, m;
ll a[2][2];
matrix(int n = 2, int m = 2): n(n), m(m) {
memset(a, 0, sizeof(a));
}
matrix operator + (const matrix &b) const {
matrix tmp(n, m);
for(int i = 0; i < n; i++) {
for(int j = 0; j < m; j++) {
tmp.a[i][j] = a[i][j] + b.a[i][j];
}}return tmp;}
matrix operator * (const matrix &b) const {
matrix tmp(n, b.m);
for(int i = 0; i < n; i++)
for(int j = 0; j < b.m; j++)
for(int k = 0; k < m; k++)
tmp.a[i][j] += a[i][k] * b.a[k][j];
return tmp;}
matrix pow(int nn) const {
matrix a = *this, tmp(n, n);
for(int i = 0; i < n; i++) {
tmp.a[i][i] = 1;}
for(; nn > 0; nn >>= 1) {
if(nn & 1) {
tmp = tmp * a;}
a = a * a;
}return tmp;}
matrix mod_mul(matrix &b, ll mod) const {
matrix tmp(n, b.m);
for(int i = 0; i < n; i++) {
for(int j = 0; j < b.m; j++) {for(int k = 0; k < m; k++)
{tmp.a[i][j] = (tmp.a[i][j] +a[i][k] * b.a[k][j] % mod) %
mod;}}}return tmp;}matrix mod_pow(ll nn, ll mod) const {
matrix a = *this, tmp(n, n);for(int i = 0; i < n; i++)
{tmp.a[i][i] = 1;}for(; nn > 0; nn >>= 1) {if(nn & 1) {tmp =
tmp.mod_mul(a, mod);}a = a.mod_mul(a, mod);}return tmp;}
}mat, ans;
```

// MERGING INTERVALS

```
sort(all(e));stack<ii> s;s.push(e[0]);int m=e.size();
rep(i,1,m){ auto
top=s.top();if(top.second<e[i].first){s.push(e[i]);}else
if(top.second<e[i].second){top.second=e[i].second;s.pop();s.
push(top);}}
```

//Sliding Window (An O(N) approach)

```
deque<pair<int,int> > window;
rep(i,1,m+1){
while(!window.empty() and window.back().f<=arr[i])
window.pop_back();
window.pb(mp(arr[i],i));
while(window.front().s<=i-b) window.pop_front();
if(i>=b) final[i][i-b+1]=window.front().f;}
```

//Fenwick Tree(Point Update and Range Query)

```
void update(lld p,lld v) {  //Add v to A[p]
for(;p<=N;p+=(p&(-p))) ft[p]+=v;
}void query(lld b) { //Sum[1….b]
lld sum = 0;for(;b>0;b-=(b&(-b))) sum+=ft[b];return sum;}
void query(lld a,lld b) {return query(b) - query(a-1);}
```

//Fenwick Tree(Range Update and Point Queries)

```
void update(lld p,lld v) {  //Add v to A[p]
for(;p<=N;p+=(p&(-p))) ft[p]+=v;
}
void update(lld a,lld b,lld v) } //Add v to A[a..b]
update(a,v);
update(b+1,-v);
}
void query(lld b) { //Value of A[b]
lld sum = 0;
for(;b>0;b-=(b&(-b))) sum+=ft[b];
return sum;
}
```

// dijkstra

```
while(size()){get = top();pop(); if(vis[get.f]) continue;
vis[get.f] = 1;for(auto j: adj[get.f]){if(dis[j.f] > get.s +
j.s){insert(j.f, dis[j.f] =  get.s + j.s);}}}
```

// SUFFIX ARRAY

//Suffix Array

```
struct SuffixArray{int L;string s;vector <vector <int> >
p;vector < pair < pair <int, int> , int > > M;
SuffixArray(string str) : L(str.length()), s(str), p(1,
vector <int> (L,0)), M(L){for(int i = 0; i < L; i++){k
p[0][i] = (int)s[i];}
for(int skip = 1, level = 1; skip < L; skip <<= 1,
level++){p.pb(vector <int> (L,0));for(int i = 0; i < L;
i++){M[i] = {{p[level-1][i], i+skip < L ? p[level-1][i+skip]
: -1},i};}
sort(M.begin(), M.end());for(int i = 0; i < L; i++){
p[level][M[i].s] = (i>0 and M[i].f==M[i-1].f ? p[level][M[i-
1].s] : i);}}}
int lcp(int i, int j){int len = 0;if(i == j) return L - i;
for(int k = p.size() - 1; k >= 0 and i < L and j < L; k--){
if(p[k][i] == p[k][j]){i+= 1<<k;j+= 1<<k;len+= 1<<k;
}}return len;}vector <int> getsa(){return p.back();
// returns index of each suffix in sorted array, take
inverse to get actual SuffArray}};
```

// A*B MOD M, A, B 10^15

(A*B-(lld)(A/(ldb)m*b+1e-3)*m+m)%m

// miller rabin

```
LL ModularMultiplication(LL a, LL b, LL m) {
LL ret=0, c=a;
while(b) {if(b&1) ret=(ret+c)%m;b>>=1; c=(c+c)%m;}
retun ret;}LL ModularExponentiation(LL a, LL n, LL m) {
LL ret=1, c=a;while(n) {
if(n&1) ret=ModularMultiplication(ret, c, m);
n>>=1; c=ModularMultiplication(c, c, m);}return ret;}
bool Witness(LL a, LL n) {LL u=n-1;int t=0;
while(!(u&1)){u>>=1; t++;}LL x0=ModularExponentiation(a, u,
n), x1;for(int i=1;i<=t;i++) {x1=ModularMultiplication(x0,
x0, n);if(x1==1 && x0!=1 && x0!=n-1) return true;x0=x1;}
if(x0!=1) return true;return false;}
```

```
LL Random(LL n) {LL ret=rand(); ret*=32768;ret+=rand();
ret*=32768;ret+=rand(); ret*=32768;ret+=rand();return ret%n;
}bool IsPrimeFast(LL n, int TRIAL) {
while(TRIAL--) {LL a=Random(n-2)+1;if(Witness(a, n)) return
false;}return true;}
```

// linkcut

```
struct Node {int sz, label; /* size, label */Node *p, *pp,
*l, *r; /* parent, path-parent, left, right pointers */
Node() { p = pp = l = r = 0; }};
void update(Node *x) {x->sz = 1;if(x->l) x->sz += x->l->sz;
if(x->r) x->sz += x->r->sz;}
void rotr(Node *x){Node *y, *z;
    y = x->p,z=y->p;
    if((y->l = x->r)) y->l->p = y;
    x->r = y, y->p = x;
    if((x->p = z))
    {   if(y == z->l) z->l = x; else z->r = x;}
    x->pp = y->pp;y->pp = 0;update(y);}
void rotl(Node *x){
Node *y, *z; y = x->p, z = y->p; if((y->r = x->l)) y->r->p =
y; x->l = y, y->p = x; if((x->p = z)){if(y == z->l) z->l =
x; else z->r = x;}x->pp = y->pp;y->pp = 0;update(y);}
void splay(Node *x){
    Node *y, *z;
    while(x->p)
    {   y = x->p;
        if(y->p == 0)
        {   if(x == y->l) rotr(x);
            else rotl(x);
        }
        else
        {   z = y->p;
            if(y == z->l)
            {   if(x == y->l) rotr(y), rotr(x);
                else rotl(x), rotr(x);
            }
            else
            {   if(x == y->r) rotl(y), rotl(x);
                else rotr(x), rotl(x);
            }
        }
    }
    update(x);
}
Node *access(Node *x){   splay(x);
if(x->r){x->r->pp = x;x->r->p = 0;x->r = 0;update(x);}
Node *last = x;while(x->pp){Node *y = x->pp;last =
y;splay(y);if(y->r){y->r->pp = y;y->r->p = 0;}y->r = x;
x->p = y;x->pp = 0;update(y);splay(x);}return last;}
Node *root(Node *x){access(x);while(x->l) x = x->l;splay(x);
return x;}
void cut(Node *x){   access(x);x->l->p = 0;x->l =
0;update(x);}
void link(Node *x, Node *y){   access(x);access(y);x->l = y;
y->p = x;update(x);}
Node *lca(Node *x, Node *y){   access(x);return access(y);}
int depth(Node *x){   access(x);return x->sz - 1;}
class LinkCut
{   Node *x;
    public:
    LinkCut(int n){   x = new Node[n];
        for(int i = 0; i < n; i++)
        {   x[i].label = i;
            update(&x[i]);
        } }
    virtual ~LinkCut(){   delete[] x;}
    void link(int u, int v){   ::link(&x[u], &x[v]);}
    void cut(int u){   ::cut(&x[u]);}
    int root(int u){   return ::root(&x[u])->label;}
    int depth(int u){   return ::depth(&x[u]);}
    int lca(int u, int v){   return ::lca(&x[u], &x[v])-
>label;  }
};
```