BIG INTEGER LIBRARY

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <deque>
#include <iomanip>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <vector>

using namespace std;

#define sd(x) x = GetNextInt()

#define LL long long
#define LD long double
#define PB push_back
#define MP make_pair
#define F first
#define S second

#define INF 2000000009
#define MAXF 5000

#if 0
#define get getchar_unlocked
#else
#define get getchar
#endif

LL next_int;
char in_char;

inline LL GetNextInt(){
    in_char = ' ';
    while((in_char < '0') | (in_char > '9')){
        in_char = get();
    }
    next_int = 0;
    while((in_char >= '0') && (in_char <= '9')){
        next_int *= 10;
        next_int += in_char - 48;
        in_char = get();
    }
    return next_int;
}


typedef pair<int,int> PII;
typedef vector<int> VI;

// base and base_digits must be consistent
const int base = 1000000000;
const int base_digits = 9;

struct bigint {
    vector<int> a;
    int sign;

    bigint() :
        sign(1) {
    }

    bigint(long long v) {
        *this = v;
    }

    bigint(const string &s) {
        read(s);
```

```
}

void operator=(const bigint &v) {
    sign = v.sign;
    a = v.a;
}

void operator=(long long v) {
    sign = 1;
    if (v < 0)
        sign = -1, v = -v;
    for (; v > 0; v = v / base)
        a.push_back(v % base);
}

bigint operator+(const bigint &v) const {
    if (sign == v.sign) {
        bigint res = v;

        for (int i = 0, carry = 0; i < (int) max(a.size(), v.a.size()) || carry; ++i) {
            if (i == (int) res.a.size())
                res.a.push_back(0);
            res.a[i] += carry + (i < (int) a.size() ? a[i] : 0);
            carry = res.a[i] >= base;
            if (carry)
                res.a[i] -= base;
        }
        return res;
    }
    return *this - (-v);
}

bigint operator-(const bigint &v) const {
    if (sign == v.sign) {
        if (abs() >= v.abs()) {
            bigint res = *this;
            for (int i = 0, carry = 0; i < (int) v.a.size() || carry; ++i) {
                res.a[i] -= carry + (i < (int) v.a.size() ? v.a[i] : 0);
                carry = res.a[i] < 0;
                if (carry)
                    res.a[i] += base;
            }
            res.trim();
            return res;
        }
        return -(v - *this);
    }
    return *this + (-v);
}

void operator*=(int v) {
    if (v < 0)
        sign = -sign, v = -v;
    for (int i = 0, carry = 0; i < (int) a.size() || carry; ++i) {
        if (i == (int) a.size())
            a.push_back(0);
        long long cur = a[i] * (long long) v + carry;
        carry = (int) (cur / base);
        a[i] = (int) (cur % base);
        //asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) : "A"(cur), "c"(base));
    }
    trim();
}

bigint operator*(int v) const {
    bigint res = *this;
    res *= v;
    return res;
}

friend pair<bigint, bigint> divmod(const bigint &a1, const bigint &b1) {
    int norm = base / (b1.a.back() + 1);
    bigint a = a1.abs() * norm;
    bigint b = b1.abs() * norm;
    bigint q, r;
    q.a.resize(a.a.size());
```

```
    for (int i = a.a.size() - 1; i >= 0; i--) {
        r *= base;
        r += a.a[i];
        int s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a.size()];
        int s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a[b.a.size() - 1];
        int d = ((long long) base * s1 + s2) / b.a.back();
        r -= b * d;
        while (r < 0)
            r += b, --d;
        q.a[i] = d;
    }

    q.sign = a1.sign * b1.sign;
    r.sign = a1.sign;
    q.trim();
    r.trim();
    return make_pair(q, r / norm);
}

bigint operator/(const bigint &v) const {
    return divmod(*this, v).first;
}

bigint operator%(const bigint &v) const {
    return divmod(*this, v).second;
}

void operator/=(int v) {
    if (v < 0)
        sign = -sign, v = -v;
    for (int i = (int) a.size() - 1, rem = 0; i >= 0; --i) {
        long long cur = a[i] + rem * (long long) base;
        a[i] = (int) (cur / v);
        rem = (int) (cur % v);
    }
    trim();
}

bigint operator/(int v) const {
    bigint res = *this;
    res /= v;
    return res;
}

int operator%(int v) const {
    if (v < 0)
        v = -v;
    int m = 0;
    for (int i = a.size() - 1; i >= 0; --i)
        m = (a[i] + m * (long long) base) % v;
    return m * sign;
}

void operator+=(const bigint &v) {
    *this = *this + v;
}
void operator-=(const bigint &v) {
    *this = *this - v;
}
void operator*=(const bigint &v) {
    *this = *this * v;
}
void operator/=(const bigint &v) {
    *this = *this / v;
}

bool operator<(const bigint &v) const {
    if (sign != v.sign)
        return sign < v.sign;
    if (a.size() != v.a.size())
        return a.size() * sign < v.a.size() * v.sign;
    for (int i = a.size() - 1; i >= 0; i--)
        if (a[i] != v.a[i])
            return a[i] * sign < v.a[i] * sign;
    return false;
}
```

```
bool operator>(const bigint &v) const {
    return v < *this;
}
bool operator<=(const bigint &v) const {
    return !(v < *this);
}
bool operator>=(const bigint &v) const {
    return !(*this < v);
}
bool operator==(const bigint &v) const {
    return !(*this < v) && !(v < *this);
}
bool operator!=(const bigint &v) const {
    return *this < v || v < *this;
}

void trim() {
    while (!a.empty() && !a.back())
        a.pop_back();
    if (a.empty())
        sign = 1;
}

bool isZero() const {
    return a.empty() || (a.size() == 1 && !a[0]);
}

bigint operator-() const {
    bigint res = *this;
    res.sign = -sign;
    return res;
}

bigint abs() const {
    bigint res = *this;
    res.sign *= res.sign;
    return res;
}

long long longValue() const {
    long long res = 0;
    for (int i = a.size() - 1; i >= 0; i--)
        res = res * base + a[i];
    return res * sign;
}

friend bigint gcd(const bigint &a, const bigint &b) {
    return b.isZero() ? a : gcd(b, a % b);
}
friend bigint lcm(const bigint &a, const bigint &b) {
    return a / gcd(a, b) * b;
}

void read(const string &s) {
    sign = 1;
    a.clear();
    int pos = 0;
    while (pos < (int) s.size() && (s[pos] == '-' || s[pos] == '+')) {
        if (s[pos] == '-')
            sign = -sign;
        ++pos;
    }
    for (int i = s.size() - 1; i >= pos; i -= base_digits) {
        int x = 0;
        for (int j = max(pos, i - base_digits + 1); j <= i; j++)
            x = x * 10 + s[j] - '0';
        a.push_back(x);
    }
    trim();
}

int length(){
   int l=0,back=a.back();
   while(back){l++;back/=10;}
   l+=((a.size()-1)*base_digits);
   return l;
}
```

```
friend istream& operator>>(istream &stream, bigint &v) {
    string s;
    stream >> s;
    v.read(s);
    return stream;
}

friend ostream& operator<<(ostream &stream, const bigint &v) {
    if (v.sign == -1)
        stream << '-';
    stream << (v.a.empty() ? 0 : v.a.back());
    for (int i = (int) v.a.size() - 2; i >= 0; --i)
        stream << setw(base_digits) << setfill('0') << v.a[i];
    return stream;
}

static vector<int> convert_base(const vector<int> &a, int old_digits, int new_digits) {
    vector<long long> p(max(old_digits, new_digits) + 1);
    p[0] = 1;
    for (int i = 1; i < (int) p.size(); i++)
        p[i] = p[i - 1] * 10;
    vector<int> res;
    long long cur = 0;
    int cur_digits = 0;
    for (int i = 0; i < (int) a.size(); i++) {
        cur += a[i] * p[cur_digits];
        cur_digits += old_digits;
        while (cur_digits >= new_digits) {
            res.push_back(int(cur % p[new_digits]));
            cur /= p[new_digits];
            cur_digits -= new_digits;
        }
    }
    res.push_back((int) cur);
    while (!res.empty() && !res.back())
        res.pop_back();
    return res;
}

typedef vector<long long> vll;

static vll karatsubaMultiply(const vll &a, const vll &b) {
    int n = a.size();
    vll res(n + n);
    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                res[i + j] += a[i] * b[j];
        return res;
    }

    int k = n >> 1;
    vll a1(a.begin(), a.begin() + k);
    vll a2(a.begin() + k, a.end());
    vll b1(b.begin(), b.begin() + k);
    vll b2(b.begin() + k, b.end());

    vll a1b1 = karatsubaMultiply(a1, b1);
    vll a2b2 = karatsubaMultiply(a2, b2);

    for (int i = 0; i < k; i++)
        a2[i] += a1[i];
    for (int i = 0; i < k; i++)
        b2[i] += b1[i];

    vll r = karatsubaMultiply(a2, b2);
    for (int i = 0; i < (int) a1b1.size(); i++)
        r[i] -= a1b1[i];
    for (int i = 0; i < (int) a2b2.size(); i++)
        r[i] -= a2b2[i];

    for (int i = 0; i < (int) r.size(); i++)
        res[i + k] += r[i];
    for (int i = 0; i < (int) a1b1.size(); i++)
        res[i] += a1b1[i];
    for (int i = 0; i < (int) a2b2.size(); i++)
```

```
            res[i + n] += a2b2[i];
        return res;
    }

    bigint operator*(const bigint &v) const {
        vector<int> a6 = convert_base(this->a, base_digits, 6);
        vector<int> b6 = convert_base(v.a, base_digits, 6);
        vll a(a6.begin(), a6.end());
        vll b(b6.begin(), b6.end());
        while (a.size() < b.size())
            a.push_back(0);
        while (b.size() < a.size())
            b.push_back(0);
        while (a.size() & (a.size() - 1))
            a.push_back(0), b.push_back(0);
        vll c = karatsubaMultiply(a, b);
        bigint res;
        res.sign = sign * v.sign;
        for (int i = 0, carry = 0; i < (int) c.size(); i++) {
            long long cur = c[i] + carry;
            res.a.push_back((int) (cur % 1000000));
            carry = (int) (cur / 1000000);
        }
        res.a = convert_base(res.a, 6, base_digits);
        res.trim();
        return res;
    }
} fib[MAXF], v;

int l, r, m;

bool pres(bigint v){
    l = 0;
    r = MAXF - 1;
    while(l <= r){
        m = (l + r) / 2;
        //cout<<l<<" "<<r<<" "<<m<<" "<<fib[m]<<endl;
        if(v == fib[m]){
            return true;
        }
        else if(v < fib[m]){
            r = m - 1;
        }
        else{
            l = m + 1;
        }
    }
    return false;
}

inline void Solve(){
    cin>>v;
    if(pres(v)){
        printf("YES\n");
    }
    else{
        printf("NO\n");
    }
}

void Pre(){
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i < MAXF; i++){
        fib[i] = fib[i - 1] + fib[i - 2];
    }
}
```

ARTICULATION POINT

```
// A C++ program to find articulation points in a given undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
       int V;  // No. of vertices
       list<int> *adj;      // A dynamic array of adjacency lists
       void APUtil(int v, bool visited[], int disc[], int low[],
                   int parent[], bool ap[]);
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // function to add an edge to graph
    void AP();    // prints articulation points
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
}

// A recursive function that find articulation points using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
// ap[] --> Store articulation points
void Graph::APUtil(int u, bool visited[], int disc[],
                                      int low[], int parent[], bool ap[])
{
    // A static variable is used for simplicity, we can avoid use of static
    // variable by passing a pointer.
    static int time = 0;

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;  // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);

            // Check if the subtree rooted with v has a connection to
            // one of the ancestors of u
            low[u]  = min(low[u], low[v]);

            // u is an articulation point in following cases

            // (1) u is root of DFS tree and has two or more chilren.
            if (parent[u] == NIL && children > 1)
```

```
                ap[u] = true;

            // (2) If u is not root and low value of one of its child is more
            // than discovery value of u.
            if (parent[u] != NIL && low[v] >= disc[u])
                ap[u] = true;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u]  = min(low[u], disc[v]);
    }
}

// The function to do DFS traversal. It uses recursive function APUtil()
void Graph::AP()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];
    bool *ap = new bool[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point) arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

    // Call the recursive helper function to find articulation points
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            APUtil(i, visited, disc, low, parent, ap);

    // Now ap[] contains articulation points, print them
    for (int i = 0; i < V; i++)
        if (ap[i] == true)
            cout << i << " ";
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nArticulation points in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.AP();
    return 0;
}
```

CONVEX HULL

```
// Implementation of Andrew's monotone chain 2D convex hull algorithm.
// Asymptotic complexity: O(n log n).
// Practical performance: 0.5-1.0 seconds for n=1000000 on a 1GHz machine.
#include <algorithm>
#include <vector>
using namespace std;

typedef int coord_t;         // coordinate type
typedef long long coord2_t;  // must be big enough to hold 2*max(|coordinate|)^2

struct Point {
        coord_t x, y;

        bool operator <(const Point &p) const {
                return x < p.x || (x == p.x && y < p.y);
        }
};

// 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
// Returns a positive value, if OAB makes a counter-clockwise turn,
// negative for clockwise turn, and zero if the points are collinear.
coord2_t cross(const Point &O, const Point &A, const Point &B)
{
        return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// Returns a list of points on the convex hull in counter-clockwise order.
// Note: the last point in the returned list is the same as the first one.
vector<Point> convex_hull(vector<Point> P)
{
        int n = P.size(), k = 0;
        vector<Point> H(2*n);

        // Sort points lexicographically
        sort(P.begin(), P.end());

        // Build lower hull
        for (int i = 0; i < n; i++) {
                while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }

        // Build upper hull
        for (int i = n-2, t = k+1; i >= 0; i--) {
                while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }

        H.resize(k);
        return H;
}
```

PRIME NUMBERS – MODIFIED SIEVE

```
#define ll long long
#define max 100000

int primes[100005];
vector<int> p;

void pre(){
	for(int i=2;i*i<=max;++i){
		if(primes[i]==0){
			int j = i*i;
			while(j<=max){
				primes[j] = 1;
				j+=i;
			}
		}
	}

	for(int i=2;i<=max;++i){
		if(primes[i]==0)
			p.push_back(i);
	}
}

void solve(int a, int b){
	int *arr;
	arr = (int*)malloc((b-a+1)*sizeof(int));

	for(int i=0;i<b-a+1;++i){
		arr[i] = 0;
	}

	int s = p.size();
	for(int i=0;(p[i]*p[i])<=b;++i){
		int f = (a-1)/p[i];
		int j = (f+1)*p[i];
		if(j==p[i])
			j+=p[i];
		while(j<=b){
			arr[j-a] = 1;
			j+=p[i];
		}
	}

	for(int i=0;i<(b-a+1);++i){
		if(a+i==1)
			continue;
		if(arr[i]==0)
			printf("%d\n", a+i);
	}
	printf("\n");
}

int main() {
	int test;
	int a, b;
	s(test);
	pre();
	while(test--){
		s(a), s(b);
		solve(a, b);
	}
	return 0;
}
```

SORT 2D ARRAY

```
int compare (const void *pa,const void *pb ) {
   /* int (*a)[3] = pa;
    int (*b)[3] = pb;
    if ( a[0][0] < b[0][0] ) return -1;
    if ( a[0][0] > b[0][0] ) return +1;
    return 0;*/

    if ( *(int*)(pa) < *(int*)(pb) ) return -1;
      if ( *(int*)(pa) > *(int*)(pb) ) return 1;
    return 0;
}

int n,a[2005][3];
for(i=0;i<n;++i){
       for(j=0;j<3;++j){
              scanf("%d",&a[i][j]);
       }
}
qsort (a, n, sizeof(a[0]), compare);
----------------------------------------

int a[100002][2];
bool myfunction (int a, int b){
       if(a>b)
              return true;
    else
              return false;
}
for(int z=0;z<t;++z)
       scanf("%d %d",&a[z][0], &a[z][1]);
sort(a[0], a[z+1], myfunction);
-----------------------------------------------------
//SORT 2-d vector
bool mySortFunction(const vector<int>& inner1, const vector<int>& inner2) {
        return inner1[1] < inner2[1];
}
vector<vector<int> > vect;
sort(vect.begin(), vect.end(), mySortFunction);
-----------------------------------------------
//SORT a structure array
struct node{
       int in;
       int val;
}a[1000];

bool compare(const node &lhs,const node &rhs)
{
     return(lhs.val<rhs.val);
}
sort(a,a+n,compare);
----------------------------------------
struct cmp_str
{
   bool operator()(char const *a, char const *b)
   {
      return std::strcmp(a, b) < 0;
   }
};

map<char *, int, cmp_str> BlahBlah;
```

CALCULATE nCr mod M

```
ll int modPow(ll int a, ll int x, ll int p) {
    //calculates a^x mod p in logarithmic time.
    ll int res = 1;
    while(x > 0) {
        if( x % 2 != 0) {
            res = (res * a) % p;
        }
        a = (a * a) % p;
        x /= 2;
    }
    return res;
}

ll int modInverse(ll int a, ll int p) {
    //calculates the modular multiplicative of a mod m.
    //(assuming p is prime).
    return modPow(a, p-2, p);
}

ll int modBinomial(ll int n, ll int k, ll int p) {
// calculates C(n,k) mod p (assuming p is prime).

    ll int numerator = 1; // n * (n-1) * ... * (n-k+1)
    for (ll int i=0; i<k; i++) {
        numerator = (numerator * (n-i) ) % p;
    }

    ll int denominator = 1; // k!
    for (ll int i=1; i<=k; i++) {
        denominator = (denominator * i) % p;
    }

    // numerator / denominator mod p.
    return ( numerator* modInverse(denominator,p) ) % p;
}
modBinomial(n,r,M);
```

BELLMAN FORD

```
void initialize(int s, int *dist, int *parent){
       for(int i=0;i<v;++i){
              dist[i] = MAX;
              parent[i] = -1;
       }
       dist[s] = 0;
}

void relax(int s, int e, int w, int *dist, int *parent){
       if(dist[e] > dist[s] + w){
              dist[e] = dist[s] + w;
              parent[e] = s;
       }
}

bool bellman_ford(int **adj, int **weight, int source){
       int *dist, *parent;
       dist = new int[v+1];
       parent = new int[v+1];

       initialize(source, dist, parent);

       for(int k=1;k<=v-1;++k){
              for(int i=0;i<v;++i){
                     for(int j=0;j<v;++j){
                            if(adj[i][j]==1){
                                   relax(i, j, weight[i][j], dist, parent);
                            }
                     }
              }
       }
}
```

MILLER – RABIN PRIMALITY TEST

```
#define X first
#define Y second
#define pb push_back
#define fr(i,n) for(int i=1;i<=n;i++)
using namespace std;
typedef long long ll;
int T;
ll mod;
int b[]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97};
ll
a[]={19960115199512061l,19951206199601151l,19951003199601151l,19510031995120 6ll,19960115199510
03ll};

ll multi(ll x,ll y)
{
	ll w=x*y-mod*(ll(double(x)*y/mod+1e-3));
	while(w<0)
		w+=mod;
	while(w>=mod)
		w-=mod;
	return w;
}
ll pow(ll x,ll y)
{
	ll t=1;
	while(y)
	{
		if(y&1)
			t=multi(t,x);
		x=multi(x,x);
		y>>=1;
	}
	return t;
}
bool judge(ll n)
{
	if(n==2) return true;
	if(n<2||!(n&1)) return false;
	for(int i=0;i<25;i++)
		if(n%b[i]==0&&n!=b[i])
			return false;
	mod=n;
	int t=0;
	ll u=n-1;
	while(!(u&1)) t++,u>>=1;
	for(int i=0;i<5;i++)
	{
		ll x=a[i]%(n-1)+1;
		x=pow(x,u);
		ll y=x;
		for(int j=1;j<=t;j++)
		{
			x=multi(x,x);
			if(x==1&&y!=1&&y!=n-1)
				return false;
			y=x;
		}
		if(x!=1) return false;
	}
	return true;
}
int main(){
		ll n;
		cin>>n;
		for(ll i=n;i>=1;i--){
			if(judge(i)){
				cout<<i<<'\n';
				break;
			}
		}
	}
	return 0;
}
```

MATRIX EXPONENTIATION – RECURSION

```
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;
typedef vector < ll > row;
typedef vector < row > matrix;

ll MOD = (ll) 1e9 + 7;

void
clear (matrix & A)
{
  for (size_t i = 0; i < A.size (); i++)
    for (size_t j = 0; j < A[i].size (); j++)
      A[i][j] = 0;
}

matrix
mul (const matrix & A, const matrix & B)
{
  matrix C = A;
  clear (C);
  for (size_t i = 0; i < C.size (); i++)
    for (size_t j = 0; j < C[i].size (); j++)
      for (size_t k = 0; k < A[i].size (); k++)
       C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % MOD;
  return C;
}

matrix
pow (const matrix & A, ll p)
{
  if (p == 0)
    {
      matrix C = A;
      clear (C);
      for (size_t i = 0; i < C.size (); i++)
       C[i][i] = 1;
      return C;
    }
  matrix C = pow (A, p / 2);
  C = mul (C, C);
  if (p & 1)
    C = mul (C, A);
  return C;
}

void
solve ()
{
  ll L;
  cin >> L;
  const int nn = 16;
  matrix Z = matrix (nn, row (nn, 0));
  matrix A = Z;
  for (int i = 1; i < nn; i++)
    A[i][i - 1] = 1;
  int k, l;
  cin >> k;
  for (int i = 0; i < k; i++)
    {
      cin >> l;
      A[0][l - 1]++;
    }
  matrix B = pow (A, L);
  cout << B[0][0] << endl;
}
```

LAZY PROPAGATION

```
/**
 * In this code we have a very large array called arr, and very large set of operations
 * Operation #1: Increment the elements within range [i, j] with value val
 * Operation #2: Get max element within range [i, j]
 * Build tree: build_tree(1, 0, N-1)
 * Update tree: update_tree(1, 0, N-1, i, j, value)
 * Query tree: query_tree(1, 0, N-1, i, j)
 */

#include <iostream>
#include <algorithm>
#include <cstdio>

using namespace std;

#include <string.h>
#include <math.h>

#define N 100005
#define inf 0x7fffffff

int arr[N];
int tree1[N];
int tree2[N];
int tree3[N];

/**
 * Build and init tree
 */
void build_tree(int node, int a, int b) {
    if(a > b) return; // Out of range

       if(a == b) { // Leaf node
       tree1[node] = arr[a]; // Init value
       tree2[node] = arr[a]; // Init value
       tree3[node] = arr[a]; // Init value
              return;
       }

       build_tree(node*2, a, (a+b)/2); // Init left child
       build_tree(node*2+1, 1+(a+b)/2, b); // Init right child

       tree1[node] = tree1[node*2]^tree1[node*2+1]; // Init root value
       tree2[node] = tree2[node*2]&tree2[node*2+1]; // Init root value
       tree3[node] = tree3[node*2]|tree3[node*2+1]; // Init root value
}

/**
 * Increment elements within range [i, j] with value value
 */
void update_tree(int node, int a, int b, int i, int j, int value) {

       if(a > b || a > j || b < i) // Current segment is not within range [i, j]
              return;

       if(a == b) { // Leaf node
       tree1[node] = value;
       return;
       }

       update_tree(node*2, a, (a+b)/2, i, j, value); // Updating left child
       update_tree(1+node*2, 1+(a+b)/2, b, i, j, value); // Updating right child

       tree1[node] = tree1[node*2]^tree1[node*2+1]; // Updating root with max value
}

/**
 * Query tree to get max element value within range [i, j]
 */
int query_tree1(int node, int a, int b, int i, int j) {

       if(a > b || a > j || b < i) return 0; // Out of range

       if(a >= i && b <= j) // Current segment is totally within range [i, j]
              return tree1[node];
```

```
	int q1 = query_tree1(node*2, a, (a+b)/2, i, j); // Query left child
	int q2 = query_tree1(1+node*2, 1+(a+b)/2, b, i, j); // Query right child

	int res = q1^q2; // Return final result

	return res;
}

int main() {
	int n, q;
	scanf("%d %d", &n, &q);

	for(int i=0;i<n;++i){
		scanf("%d", &arr[i]);
	}

	build_tree(1, 0, n-1);

	while(q--){
		int action;
		scanf("%d", &action);
		if(action==1){
			int val, index;
			scanf("%d %d", &val, &index);
			update_tree(1, 0, n-1, index-1, index-1, val); // Increment range [0, 6]
by 5
		} else {
			char option[10];
			int l, r;
			scanf("%s %d %d", option, &l, &r);
			if(option[0]=='X'){
				cout << query_tree1(1, 0, n-1, l-1, r-1) << endl; // Get max
element in range [0, N-1]*/
			}
		}
	}
}
```

FLOYD WARSHALL

```
1 let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
2 for each vertex v
3    dist[v][v] ← 0
4 for each edge (u,v)
5    dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
6 for k from 1 to |V|
7    for i from 1 to |V|
8       for j from 1 to |V|
9          if dist[i][j] > dist[i][k] + dist[k][j]
10             dist[i][j] ← dist[i][k] + dist[k][j]
11         end if
```

INVERSION COUNT

```
/* This function sorts the input array and returns the
   number of inversions in the array */
int mergeSort(int arr[], int array_size)
{
    int *temp = (int *)malloc(sizeof(int)*array_size);
    return _mergeSort(arr, temp, 0, array_size - 1);
}

/* An auxiliary recursive function that sorts the input array and
  returns the number of inversions in the array. */
int _mergeSort(int arr[], int temp[], int left, int right)
{
  int mid, inv_count = 0;
  if (right > left)
  {
    /* Divide the array into two parts and call _mergeSortAndCountInv()
       for each of the parts */
    mid = (right + left)/2;

    /* Inversion count will be sum of inversions in left-part, right-part
      and number of inversions in merging */
    inv_count  = _mergeSort(arr, temp, left, mid);
    inv_count += _mergeSort(arr, temp, mid+1, right);

    /*Merge the two parts*/
    inv_count += merge(arr, temp, left, mid+1, right);
  }
  return inv_count;
}

/* This funt merges two sorted arrays and returns inversion count in
   the arrays.*/
int merge(int arr[], int temp[], int left, int mid, int right)
{
  int i, j, k;
  int inv_count = 0;

  i = left; /* i is index for left subarray*/
  j = mid;  /* i is index for right subarray*/
  k = left; /* i is index for resultant merged subarray*/
  while ((i <= mid - 1) && (j <= right))
  {
    if (arr[i] <= arr[j])
    {
      temp[k++] = arr[i++];
    }
    else
    {
      temp[k++] = arr[j++];

     /*this is tricky -- see above explanation/diagram for merge()*/
      inv_count = inv_count + (mid - i);
    }
  }

  /* Copy the remaining elements of left subarray
   (if there are any) to temp*/
  while (i <= mid - 1)
    temp[k++] = arr[i++];

  /* Copy the remaining elements of right subarray
   (if there are any) to temp*/
  while (j <= right)
    temp[k++] = arr[j++];

  /*Copy back the merged elements to original array*/
  for (i=left; i <= right; i++)
    arr[i] = temp[i];

  return inv_count;
}
```