Algoritmos numericos

Inverso modular

```
long long mod_inverse(long long a, long long n) {
   long long i = n, v = 0, d = 1;
   while(a > 0) {
      long long t = i/a, x = a;
      a = i%x;
      i = x;
      x = d;
      d = v - t*x;
      v = x;
   }
   v %= n;
   if(v < 0)
      v += n;
   return v; // (a * v) mod n = 1
}
```

Teorema Chino del Residuo

```
long long chinese_reminder(int len, long long B) {
   long long x = 0;
   //vector<long long> m(len), c(len);
   for(int i = 0; i < len; i++) {
      long long m = B/b[i];
      x += (a[i] * m * mod_inverse(m, b[i])) % B;
      //En caso de ser necesarios los vectores m[i] y c[i]
      // m[i] = B/b[i];
      // c[i] = m[i] * mod_inverse(m[i], b[i]);
      // x += (a[i] * c[i]) % B;
   }
   return x%B;
}
```

Seno y Coseno

```
const long double pi = 3.141592654;

/* Como calcular el seno de un ángulo utilizando
  la serie de Taylor */
long double ssin(long double x) {
   int n = 20;
   long double sign = 1, factorial = 1, xp = x;
   long double ans = x;
   for(int i=1; i<=n; i++){
      sign *= -1;
      factorial *= (2*i) * (2*i + 1);
      xp *= x * x;
      ans += (sign * xp) / factorial;
   }
   return ans;
}

/* Como calcular el coseno de un ángulo utilizando
  la serie de Taylor */
long double ccos(long double x) {
   int n = 20;
   long double sign = 1, factorial = 1, xp = 1;
   long double ans = 1;
   for(int i = 1; i <= n; i++) {
      sign *= -1;
```

```
    factorial *= (2*i - 1) * (2*i);
    xp *= x * x;
    ans += (sign * xp) / factorial;
  }
  return ans;
}
```

Números Random

```
//Esta función genera numeros aleatorios hasta de 64 bits
mt19937_64
rng(chrono::steady_clock::now().time_since_epoch().count());

int main (){
  long long x;
  x = rng();
  cout<<x<<endl;
}
```

Números Practicos

```
set<long long> getDivisors(long long n) {
  set<long long> divisors;
  for (long long i = 1; i * i <= n; ++i)
    if (n % i == 0)
      divisors.insert({i, n / i});
  return divisors;
}

bool isPractical (long long n) {
  set<long long> divisors = getDivisors(n);
  long long sum = 0;
  for (long long divisor : divisors) {
    if (divisor > sum + 1)
      return false;
    sum += divisor;
  }
  return true;
}

int main() {
  long long m;
  int t; cin >> t;
  while (t--) {
    cin >> m;
    cout << (isPractical(m) ? "Yes" : "No") << endl;
  }
}

/* Cabe resaltar que todos los multiplos de 6 son número prácticos,
además
de que el único impar que es número práctico es el 1 */
```

GCD y LCM

```
long long gcd (long long A, long long B) {
  long long temp;
  while (b != 0) {
    temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

```
ll lcm(ll a, ll b){
   return (a / gcd(a, b)) * b;
}
```

Exponenciación Binaria

```
ll myPow(ll b, ll e){
   ll result = 1;
   while (e > 0) {
      if (e % 2 == 1)
         result = ((b % mod) * (result % mod)) % mod;
      b = ((b % mod) * (b % mod)) % mod;
      e /= 2;
   }
   return result;
}
```

Criba de Erastótenes

```
#define limite 10001000
int main() {
   bool es_primo[limite + 1];

   es_primo[0] = es_primo[1] = false;
   for (int i = 2; i <= limite; ++i)
      es_primo[i] = true;

   for (long long int p = 2; p * p <= limite; p++) {
      if (es_primo[p]) {
         for (long long int i = p * p; i <= limite; i += p)
            es_primo[i] = false;
      }
   }
   return 0;
}
```

SQRT

Segment Tree

Maximo (no lazy)

```
const int MAXT = 100000;
const int myNegativeInfinite = -2e9;

vector<int> tree(MAXT*2 + 1);
int n;

void buildTree(vector<int>& A) {
   for (int i = 0; i < n; i++)tree[i + n] = A[i];
   for (int i = n - 1; i > 0; --i)tree[i] = max(tree[i << 1], tree[(i << 1) ^
1]); //i << 1 -> i*2
   // Si i << 1 es el hijo izquierdo, (i << 1) ^ 1 es el hijo derecho y
viceversa
}

void updateTree(int update, int value) {
   tree[n + update] = value;
   update += n;
   for (int i = update; i > 1; i >>= 1) tree[i>>1] = max(tree[i], tree[i ^
1]); //Si i es el hijo por izquierda de i/2, entonces i ^ 1 es el hijo por
derecha, y viceversa
   //i >> i -> i/2
}
```

```
int query(int l, int r) {
   int ans = myNegativeInfinite;
   l += n; r += n;
   while (l < r) {
      if (l & 1)ans = max(ans, tree[l++]);
      if (r & 1)ans = max(ans, tree[--r]);
      l >>= 1; r >>= 1;
   }

   return ans;
}

int main() {
   ios_base::sync_with_stdio(false);cin.tie(NULL);
   int q; cin >> n >> q;
   vector<int> A; int save;
   for (int i = 0; i < n; i++) {
      cin >> save;
      A.push_back(save);
   }
   buildTree(A);

   int op;
   while (q--) {
      cin >> op;
      if (op == 1) {
         int x, w;cin >> x >> w;
         updateTree(x-1, tree[n+x-1]+w);
      }
      else {
         int l, r;cin >> l >> r;
         cout << query(l-1, r) << endl; //La implementacion es [l, r), entonces a r no le resto 1 porque en el ejercicio si necesito incluirlo
      }
   }

   return 0;
}
```

Suma (Lazy)

```
struct SegTree {
   int n;
   vector<int> tree, setLazy, begin, end;
   void prop(int i) {
      if (setLazy[i] != -100) {
         setLazy[2 * i + 1] = setLazy[2 * i] = setLazy[i];
         tree[2 * i] = tree[2 * i + 1] =
            setLazy[i] * (end[2 * i + 1] - begin[2 * i + 1] + 1);
         setLazy[i] = -100;
      }
   }
   SegTree(int nn) {
      n = 1;
      while (n < nn) n *= 2;
      tree.resize(2 * n);
      setLazy.resize(2 * n, -100);
      begin.resize(2 * n);
      end.resize(2 * n);
      for (int i = n; i < 2 * n; i++) {
         begin[i] = end[i] = i - n;
      }
      for (int i = n - 1; i > 0; i--) {
         begin[i] = begin[2 * i];
```

```
        end[i] = end[2 * i + 1];
      }
    }
    void rangeSet(int i, int amt, int lo, int hi) {
      if (i < n) prop(i);
      if (begin[i] == lo && end[i] == hi) {
        tree[i] = amt * (hi - lo + 1);
        setLazy[i] = amt;
        return;
      }
      if (begin[2 * i] <= hi && end[2 * i] >= lo) {
        rangeSet(2 * i, amt, lo, min(hi, end[2 * i]));
      }
      if (begin[2 * i + 1] <= hi && end[2 * i + 1] >= lo) {
        rangeSet(2 * i + 1, amt, max(lo, begin[2 * i + 1]), hi);
      }
      tree[i] = tree[2 * i] + tree[2 * i + 1];
    }
    int query(int i, int lo, int hi) {
      if (i < n) prop(i);
      if (begin[i] == lo && end[i] == hi) return tree[i];
      int ans = 0;
      if (begin[2 * i] <= hi && end[2 * i] >= lo) {
        ans += query(2 * i, lo, min(end[2 * i], hi));
      }
      if (begin[2 * i + 1] <= hi && end[2 * i + 1] >= lo) {
        ans += query(2 * i + 1, max(lo, begin[2 * i + 1]), hi);
      }
      return ans;
    }
};
```

```
int main (){
    ios_base::sync_with_stdio(0);cin.tie(NULL);

    int n;cin>>n;
    vector<int> a(n);
    for(int i = 0; i < n; i++)
        cin>>a[i];

    //Inicializacion del Segment Tree
    SegTree st(n);

    //Actualizaciones (el primer valor siempre debe ser 1)
    st.rangeSet(1, a[0], 1, 5); /*En este caso pondriamos el valor de
a[0] en
                                cada uno de los elementos del array del
                                segment tree desde 1 hast 5*/

    //Para actualizar un solo valor
    st.rangeSet(1, 5, 3, 3); /*El limite inferior y superior son el mismo,
entonces
                            solo se actualiza esa posicion*/

    //Query (el primer valor siempre debe ser 1)
    int ans = st.query(1, 2, 6); /*Suma desde el elemento 2 hasta el 6
del array del
                                segment tree*/

    cout<<ans<<endl;

    return 0;
}
```

Sparse Table

Maximo

```
// Si bien en este ejemplo se usa 1-index, el algoritmo es valido
// para 0-index, simplemente se deben cambiar los limites de
// i dentro del for(i = 1; i <= n; i++) por for(i = 0; i < n; i++)
// tanto en la lectura de los datos como en el llenado de la sparse
// table.

const int lgn = 17; //Piso del Lg2 de MAXN (maximo numero de
elementos)
const ll MAXN = 2e5 + 2;
typedef pair<ll, int> pli;

vector<ll> a(MAXN);
pli rmq[MAXN][lgn + 1];

void solver(){
   int n;cin>>n;

   for(int i = 1; i <= n; i++){
      cin>>a[i];
      rmq[i][0] = {a[i], i}; //Inicializar sparse table
   }

   //Llenar sparse table
   for (int j = 1; j <= lgn; j++) {
      for (int i = 1; i + (1 << j) - 1 <= n; i++)
         rmq[i][j] = max(rmq[i][j - 1], rmq[i + (1 << (j - 1))][j - 1]);
   }

   //Queries
   int q; cin>>q;
   while(q--){
      int l, r; cin>>l>>r;
      int lg = 31 - __builtin_clz(r - l + 1); //Piso del Log2 del tamaño
del rango
      pli maxi = max(rmq[l][lg], rmq[r - (1 << lg) + 1][lg]); //[l, r]
      cout<<"El mayor numero entre "<<l<<" y "<<r<<" es:
"<<maxi.first<<", y se encuentra en la posicion:
"<<maxi.second<<endl;
   }
}
```

Grafos

BFS

```
void BFS(int source, int n, vector<int>& distance, vector<int>&
parent){
   vector<int> visited(n+1, 0);
   for(int i = 1; i <= n; i++) {
      distance[i] = inf;
      parent[i] = -1;
   }
   distance[source] = 0;
   queue<int> q;
   q.push(source);
   visited[source] = 1;

   while(!q.empty()){
      int u = q.front();
      q.pop();
      for(int v : graph[u]){
```

```
        if(visited[v] == 0){
            visited[v] = 1;
            distance[v] = distance[u] + 1;
            parent[v] = u;
            q.push(v);
        }
    }
  }
}
```

DFS

```
void DFS(int u, vector<int>& visited, vector<int>& parent) {
  visited[u] = 1;
  for(int v : graph[u]) {
    if(!visited[v]) {
      parent[v] = u;
      DFS(v, visited, parent);
    }
  }
}
```

Dijkstra

```
struct comp{
  bool operator() (pii a, pii b){
    return a.second > b.second;
  }
};

void dijkstra(int source, int n, vector<int>& distance, vector<int>&
parent){
  for(int i = 1; i <= n; i++){
    distance[i] = inf;
    parent[i] = -1;
  }
  distance[source] = 0;
  priority_queue<pii, vector<pii>, comp> pq;
  pq.push({source, 0});
  while(!pq.empty()){
    int u = pq.top().first;
    pq.pop();
    for(pii v : graph[u]){
      if(distance[v.first] > distance[u] + v.second){
        distance[v.first] = distance[u] + v.second;
        parent[v.first] = u;
        pq.push({v.first, distance[v.first]});
      }
    }
  }
}
```

Prim

```
void prim(int source, int n, vector<int>& distance, vector<int>&
parent){
  for(int i = 1; i <= n; i++){
    distance[i] = inf;
    parent[i] = -1;
  }
  distance[source] = 0;
  priority_queue<pii, vector<pii>, comp> pq;
  pq.push({source, 0});
  while(!pq.empty()){
    int u = pq.top().first;
    pq.pop();
    for(pii v : graph[u]){
```

```
        if(distance[v.first] > v.second){
          distance[v.first] = v.second;
          parent[v.first] = u;
          pq.push({v.first, distance[v.first]});
        }
      }
   }
}
```

Dijstra variante pesos negativos

```
struct comparator{
        bool operator() (pii a, pii b){
                return a.second > b.second;
        }
};

const int maxN = 5000;
vector<pii> graph[maxN+1];
vector<int> dist(maxN);

void dijkstra(int n, int z){
        priority_queue<pii, vector<pii>, comparator> pq;
        vector<int> S(n+1);
        for(int x=1; x<=n; x++){
                dist[x] = inf;
                S[x] = 0;
        }
        dist[z] = 0;

        pq.push(pii(z,0));
        while(!pq.empty()){
                pii p = pq.top(); pq.pop();
                int x = p.first;
                S[x] = 1;
                for(int i=0; i<graph[x].size(); i++){
                        int y = graph[x][i].first;
                        int w = graph[x][i].second;
                        if(dist[x]+w < dist[y]){
                                dist[y] = dist[x] + w;
                                pq.push(pii(y,w));
                        }
                }
        }
}
```

Generate Tree

```
/* Esta funcion sirve para cuando tenemos grafo tipo
arbol y queremos que se almacene como tal */

const int maxN = 1000000;
vector<int> graph[maxN+1];
vector<int> tree[maxN+1];

void generateTree(int root){
   queue<int> Q;
   vector<int> S(maxN+1, 0);
   Q.push(root); S[root] = 1;
   while(!Q.empty()){
      int x = Q.front(); Q.pop();
      for(int i=0; i<graph[x].size(); i++){
         int y = graph[x][i];
         if(S[y] == 0){
            S[y] = 1;
            Q.push(y);
```

```
            tree[x].push_back(y);
        }
    }
  }
}
```

Comprobar si un grafo es bipartito

```
void DFS(int u) {
  for (auto v: g[u]) {
    if (!color[v]) {
      color[v] = 3 - color[u];
      DFS(v);
    }
    else if (color[v] + color[u] != 3)
      bipartito = false;
  }
}
```

Potencia de una matriz

```
//Multiplicación de matrices
matrix multMatrix (const matrix &A, const matrix &B) {
  int len = A.size();
  matrix result(len, vector<long long>(len, 0));
  for (int i = 0; i < len; i++) {
    for (int j = 0; j < len; j++) {
      for (int k = 0; k < len; k++)
        result[i][j] = (result[i][j] + A[i][k] * B[k][j]) % mod;
    }
  }
  return result;
}
```

```
//Potencia de una matriz
matrix potMatrix (matrix A, long long expo) {
  int len = A.size();
  matrix result(len, vector<long long>(len, 0));
  for (int i = 0; i < len; i++)
    result[i][i] = 1;
  while (expo > 0) {
    if (expo % 2 == 1)
      result = multMatrix(result, A);
    A = multMatrix(A, A);
    expo /= 2;
  }
  return result;
}
//La matriz A representa la matriz de adyacencia del grafo
//La matriz B representa la matriz de adyacencia del grafo elevada a
la n-1
//La potencia de la matriz de adyacencia del grafo es la cantidad de
caminos de longitud n entre los vertices
//Si quiero saber el número de caminos de longitud n, debo elevar la
matriz de adyacencia a la n-1
```

LCA

```
/* Este algoritmo sirve para calcular el minimo ancestro
  comun (Lowest Common Ancestor) entre dos nodos especificos.
  Cada query tiene una complejidad de 0(lg(n)), siendo n
  el número de nodos del grafo, gracias al Binary Lifting.
  El algoritmo toma un nodo en especifico como raiz (en este
  caso el nodo 0), y calcula la profundidad de cada nodo
  según el número minimo de nodos que debe atravesar para
  llegar al nodo raíz, esto se calcula mediante un BFS. */
```

```
// Esta implementación sirve para grafos no dirigidos

const int MAXT = 1001;
const int LOG = 12; //El logaritmo base 2 del numero maximo de
nodos
vector<vector<int>> graph(MAXT, vector<int>()), up(MAXT,
vector<int>(LOG, 0));
vector<int> depth(MAXT, -1);
priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;

void BFS () {
   depth[0] = 0;
   int v;
   pq.push({0, 0});
   while(!pq.empty()) {
      v = pq.top().second;
      pq.pop();
      for(int u : graph[v]) {
         if(depth[u] == -1) {
            depth[u] = depth[v] + 1;
            up[u][0] = v;

            for(int j = 1; j < LOG; j++)
               up[u][j] = up[up[u][j-1]][j-1];
            pq.push({depth[u], u});
         }
      }
   }
}

int LCA (int a, int b) {
   if(depth[a] < depth[b])
      swap(a, b);
   int k = depth[a] - depth[b];
   for(int j = LOG - 1; j >= 0; j--) {
      if(k & (1 << j))
         a = up[a][j];
   }
   if(a == b)
      return a;
   for(int j = LOG - 1; j >= 0; j--) {
      if(up[a][j] != up[b][j]) {
         a = up[a][j];
         b = up[b][j];
      }
   }
   return up[a][0];
}

int main(){
   int n, m;
   cin>>n>>m;
   for(int i = 0; i < m; i++){
      int u, v;
      cin>>u>>v;
      graph[u].push_back(v);
      graph[v].push_back(u);
   }

   /* Asumiendo que es un grafo conectado y que nuestro
      nodo raíz es 0, vamos a calcular el LCA de cada par
      de nodos del grafo */
```

```
   for(int i = 0; i < n; i++){
      for(int j = i+1; j <= n; j++)
         cout<<"El Lowest Common Ancestor entre "<<i<<" y "<<j<<"
es: "<<LCA(i, j)<<endl;
   }

   return 0;
}
```

MaxFlow (Dinic)

```
// Aplicado en grafos de flujo dirigidos con capacidades en las aristas
// Complejidad O(v^2 * E)
// Para grafos con capacidades unitarias o en redes densas O(sqrt(V)
* E)

/* Otra aplicacion: Minimum vertex cut
   Para hallar el minimum vertex cut tendriamos que llamar Dinic con
2*n + 2
   y añadir una arista entre cada 2*i y 2*i + 1 con capacidad de 1,
para
   0 <= i < n. Despues se agregan aristas con capacidad 1 según las
condiciones
   del ejercicio o del grafo original. Si buscamos añadir una arista
desde un
   vertice u hasta un vertice v, usamos: graph.addEdge(2*u + 1, 2*v,
1).
   Despues agregamos aristas desde el nodo 2*n hacia todos los
nodos iniciales
   y desde todos los nodos finales hacia el nodo 2*n + 1.
   Finalmente, usamos graph.calc(2*n, 2*n + 1), para hallar el
maximo flujo
   desde 2*n (que cumple la funcion de source-fuente), hasta 2*n + 1
(que cumple
   la funcion de sink-sumidero). El valor del maximo flujo
corresponde al valor
   del minimum vertex cut. */

#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
using ll = long long;

typedef pair<int, int> pii;

struct comp{
   bool operator() (pii a, pii b){
      if(a.second == b.second)
         return a.first < b.first;
      return a.second < b.second;
   }
};

struct Dinic {
   struct Edge {
      int to, rev;
      ll c, oc;
      ll flow() { return max(oc - c, 0LL); } // if you need flows
   };
   vector<int> lvl, ptr, q;
   vector<vector<Edge>> adj;
   Dinic(int n) : lvl(n), ptr(n), q(n), adj(n) {}
   void addEdge(int a, int b, ll c, ll rcap = 0) {
      adj[a].push_back({b, adj[b].size(), c, c});
```

```
    adj[b].push_back({a, adj[a].size() - 1, rcap, rcap});
  }
  ll dfs(int v, int t, ll f) {
    if (v == t || !f) return f;
    for (int& i = ptr[v]; i < adj[v].size(); i++) {
      Edge& e = adj[v][i];
      if (lvl[e.to] == lvl[v] + 1)
        if (ll p = dfs(e.to, t, min(f, e.c))) {
          e.c -= p, adj[e.to][e.rev].c += p;
          return p;
        }
    }
    return 0;
  }
  ll calc(int s, int t) {
    ll flow = 0; q[0] = s;
    for(ll L = 0; L < 31; L++) do { // 'int L=30' maybe faster for
random data
      lvl = ptr = vector<int> (q.size());
      int qi = 0, qe = lvl[s] = 1;
      while (qi < qe && !lvl[t]) {
        int v = q[qi++];
        for (Edge e : adj[v])
          if (!lvl[e.to] && e.c >> (30 - L))
            q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
      }
      while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
    } while (lvl[t]);
    return flow;
  }
  bool leftOfMinCut(int a) { return lvl[a] != 0; }
};
```

```
int main() {
  ios_base::sync_with_stdio(0);cin.tie(NULL);
  int n; cin>>n;
  Dinic graph(n + 2); //Crear un grafo para usar Dinic

  graph.addEdge(s, d, c); //Agregar una arista desde s hasta d con
capacidad c

  graph.calc(source, sink); //Maximo flujo desde source hast sink

  return 0;
}
```

Otros

Inversiones

```
#define myInfinite 2147483647
#define MAXT 100

long long int inv = 0;

void myMerge (vector<int>& A, int p, int q, int r) {
  int n1 = q-p+1, n2 = r-q, i, j, k;
  vector<int> L(n1+2), R(n2+2);
  for (i=1; i<=n1; i++)
    L[i] = A[p+i-1];
  for (j=1; j<=n2; j++)
    R[j] = A[q+j];
  L[n1+1] = myInfinite;
  R[n2+1] = myInfinite;
```

```
   i=1;
   j=1;
   for (k=p; k<=r; k++) {
      if (L[i] <= R[j]) {
         A[k] = L [i];
         i++;
      }
      else {
         A[k] = R[j];
         j++;
         inv += n1 - i + 1;
      }
   }
}

void MergeSort (vector<int>& A, int p, int r) {
   int q;
   if (p<r) {
      q = (p+r)/2;
      MergeSort (A, p, q);
      MergeSort (A, q+1, r);
      myMerge (A, p, q, r);
   }
}
```

Hashing XOR

/* El hashing map sirve para almacenar elementos en un conjunto especifico. Consiste en leer un numero a para almacenar cierto dato en un vector, entonces al número que tenemos en la posicion a de nuestro vector le hacemos xor con un número r random, asegurando de esta manera que el valor que tendrá será único */

/* El ejercicio que se encuentra en este archivo consiste en t casos de prueba. Nos dan n ciudades y todas están conectadas de la forma i con i+1, para i < n, y para i = n se dice que está conectada con la ciudad 1. Entonces nos dan m pares de amigos que pertenecen respectivamente a la ciudad a y a la ciudad b. Entonces se nos pregunta cual es la minima cantidad de caminos que puede tener la ciudad de modo que cada par de amigos esté conectado.
Para resolver este ejercicio usamos hashing xor de la siguiente manera:
Se leen los número a y b y al valor que tenía en cada posición se le hace un XOR con un número random (de 64 bits para evitar colisiones).
Posteriormente, queremos encontrar un número fixAns que sea la mayor cantidad de números consecutivos iguales, de modo que podamos decir que puedo conectar todas las ciudades usando n - fixAns caminos. Debido a que nuestro vector tiene guardados los números a los que les aplicamos XOR, podemos elegir un número arbitrario save que guarde en dicha posición dentro de un mapa las veces que encontré save. Save va a cambiar según el número que estamos evaluando del vector y solo volverá a ser el mismo número que en una iteración anterior cuando cruce por un número en el vector que sea diferente de 0, o cuando halla pasado por el mayor número a o b que leí al inicio del ejercicio.*/

/* Link del ejercicio:
https://codeforces.com/problemset/problem/1996/G
Para una explicación más detallada de la logica detrás de está colución, dirigirse a la editorial, la cual se puede encontrar en el link adjunto*/

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

```
#define endl '\n'

// "rng" genera un número random de 64 bits
mt19937_64
rng(chrono::steady_clock::now().time_since_epoch().count());

void solver() {
   int n, m, a, b;
   long long r;
   cin>>n>>m;
   vector<long long> D(n);
   while(m--) {
      cin>>a>>b;
      r = rng();
      D[a-1] ^= r;
      D[b-1] ^= r;
   }

   unordered_map<long long, int> conjuntos;
   int fixAns = 0;
   long long save = 0;
   for(long long diag : D)
      fixAns = max(fixAns, conjuntos[save^=diag]++);

   cout<<n-fixAns-1<<endl;
}

int main() {
   ios_base::sync_with_stdio(0);cin.tie(NULL);
   int t;cin>>t;
   while(t--) {
      solver();
   }

   return 0;
}
```