```
// Hopcroft – Karp O(M*sqrt(N))
const int MAXV = 1001;
const int MAXV1 = 2*MAXV;
int N, M;
vector<int> ady[MAXV];
int D[MAXV1], Mx[MAXV], My[MAXV];

bool BFS(){
  int u, v, i, e;
  queue<int> cola;
  bool f = 0;
  for (i = 0; i < N+M; i++) D[i] = 0;
  for (i = 0; i < N; i++)
  if (Mx[i] == -1) cola.push(i);
  while (!cola.empty()) {
    u = cola.front(); cola.pop();
    for (e = ady[u].size()-1; e >= 0; e--) {
      v = ady[u][e];
      if (D[v + N]) continue;
      D[v + N] = D[u] + 1;
      if (My[v] != -1) {
        D[My[v]] = D[v + N] + 1;
        cola.push(My[v]);
      }
      else f = 1;
    }
  }
  return f;
}

int DFS(int u) {
  for (int v, e = ady[u].size()-1; e >=0; e--) {
    v = ady[u][e];
    if (D[v+N] != D[u]+1) continue;
    D[v+N] = 0;
    if (My[v] == -1 || DFS(My[v])) {
      Mx[u] = v; My[v] = u;
      return 1;
    }
  }
  return 0;
}

int Hopcroft_Karp(){
  int i, flow = 0;
  for (i = max(N,M); i >=0; i--)
    Mx[i] = My[i] = -1;
  while (BFS())
    for (i = 0; i < N; i++)
      if (Mx[i] == -1 && DFS(i))
        ++flow;
  return flow;
}
```

```cpp
// Heavy Light Descomposition
int N, M;
vector<int> V[MN];
vector<int> G[MN];
vector<bool> L[MN];
/// cant- la cantidad de nodos
/// pos- la pos. donde aparece
/// nn- el nod en el cual aparece
/// pd- el link con el padre full superior
/// G-Dp
/// L-lazy
int cant[MN], pos[MN], nn[MN], pd[MN];

void Dfs(int nod, int pad){
  int t = V[nod].size(), newn;
  if (t == 1 && nod != 1){
    pos[nod] = 0;
    nn[nod] = nod;
    cant[nod] = 1;
    pd[nod] = pad;
    return;
  }
  int mej = nod;
  for (int i = 0; i < t; i ++){
    newn = V[nod][i];
    if (newn == pad) continue;
    Dfs (newn, nod);
    if (cant[mej] < cant[nn[newn]])
      mej = nn[newn];
  }
  pos[nod] = cant[mej];
  cant[mej] ++;
  nn[nod] = mej;
  pd[mej] = pad;
}

typedef pair<int, int> par;
typedef pair<int, par> tri;
typedef vector<tri> vt;
typedef vector<par> vp;
/// me da el recorrido desde a hasta b en vector<tri>
/// f posicion s.f in, s.f fin
```

```cpp
vt rec(int a, int b) {
  vp A1, B1;
  A1.clear(), B1.clear();
  for (int i = a; i != -1; i = pd[nn[i]])
    A1.push_back(par(nn[i], pos[i]));
  for (int i = b; i != -1; i = pd[nn[i]])
    B1.push_back(par(nn[i], pos[i]));
  vt C1;
  C1.clear();
  reverse(A1.begin(), A1.end());
  reverse(B1.begin(), B1.end());
  int t = 0;
  while (t < A1.size() && t < B1.size() && A1[t] == B1[t])
    t ++;
  if (t >= A1.size() || t >= B1.size() || (t < B1.size() &&
  t < A1.size() && A1[t].first != B1[t].first))
    t --;
  if ((t < A1.size() && t < B1.size()) && A1[t].first ==
    B1[t].first){
    C1.push_back(tri(A1[t].first, par(min(A1[t].second,
    B1[t].second), max(A1[t].second, B1[t].second))));
    t ++;
  }
  for (int i = t; i < A1.size(); i ++)
    C1.push_back (tri(A1[i].first, par(A1[i].second,
    cant[A1[i].first] - 1)));
  for( int i = t; i < B1.size(); i ++)
    C1.push_back (tri(B1[i].first, par(B1[i].second,
    cant[B1[i].first] - 1)));
  return C1;
}

void havy_light() {
  Dfs (1, -1); // root
  for (int i = 1; i <= N; i ++) /// rellenar con 4*cant
    if(cant[i]){
      G[i] = vector<int> (cant[i]*4, 0);
      L[i] = vector<bool> (cant[i]*4, false);
      G[i][1] = cant[i], L[i][1] = true;
    }
}
```

```cpp
// Segment Tree Persistente
const int N = 100000 + 100, LOGN = 20;
const int TOT = 4*N + N*LOGN;
int sum[TOT], L[TOT], R[TOT];
int sz = 1;

int newNode(int s = 0){
  sum[sz] = s;
  return sz++;
}

int build(int b, int e){
  if (b==e) return newNode();
  int mid = (b + e) >> 1;
  int cur = newNode();
  L[cur] = build(b, mid);
  R[cur] = build(mid+1 , e);
  return cur;
}

int update(int node, int b, int e, int p){
  if(b == e) return newNode(sum[node] + 1);
  int mid = (b + e) >> 1;
  int cur = newNode();
  if(p <= mid) {
    L[cur] = update(L[node], b, mid, p);
    R[cur] = R[node];
  }
  else {
    R[cur] = update(R[node], mid+1 , e, p);
    L[cur] = L[node];
  }
  sum[cur] = sum[L[cur]] + sum[R[cur]];
  return cur;
}

int query(int node1, int node2, int b, int e, int k){
  if(b == e) return b;
  int s = sum[L[node2]] - sum[L[node1]];
  int mid = (b + e) >> 1;
  if(s >= k) return query(L[node1], L[node2], b, mid, k);
  else return query(R[node1], R[node2], mid+1 , e, k-s);
}

int root[N];

int main()
{
  int n, m;
  cin >> n >> m;
  root[0] = build(1 , n);
  vector<int> v(n), tmp(n);
  for(int i = 0; i < n; ++i){
    cin >> v[i]; tmp[i] = v[i];
  }
  sort(tmp.begin(), tmp.end());
  tmp.resize(unique(tmp.begin(), tmp.end()) - tmp.begin());
  for(int i = 0; i < n; ++i)
    root[i+1] = update(root[i], 1 , n, lower_bound (
    tmp.begin(), tmp.end(), v[i]) - tmp.begin() + 1);
  while(m--){
    int i, j, k;
    cin >> i >> j >> k;
    cout << tmp[query(root[i-1], root[j], 1 , n, k)-1];
  }
}
```

```
// Pollard's Rho Integer Factoring Algorithm
typedef long long ll;

ll mulmod(ll a, ll b, ll c) { // (a*b)%c, minimizing overflow
  ll x = 0, y = a % c;
  while (b > 0) {
    if (b % 2 == 1)
      x = (x + y) % c;
    y = (y * 2) % c;
    b /= 2;
  }
  return x % c;
}

ll pollard_rho(ll n) {
  int i = 0, k = 2;
  ll x = 3, y = 3; // random seed = 3, other values possible
  while (1) {
    i++;
    x = (mulmod(x, x, n) + n - 1) % n;
    ll d = __gcd(abs(y - x), n);
    if (d != 1 && d != n) return d;
    if (i == k) y = x, k *= 2;
  }
}

int main() {
  ll n = 2063512844981574047LL; // n is not a large prime
  ll ans = pollard_rho(n);
  if (ans > n / ans) ans = n / ans;
  cout << ans << ' ' << n / ans;
  // should be: 1112041493 1855607779
  return 0;
}
```

```
// Floyd’s Cycle-Finding Algorithm O(μ + λ) μ->mu, λ->lambda
typedef pair<int,int> ii;
int f(int x);

ii floydCycleFinding(int x0) {
  // finding k*mu, hare’s speed is 2x tortoise’s
  int tortoise = f(x0);
  int hare = f(f(x0)); // f(x0) is the node next to x0
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(f(hare));
  }
  // finding mu, hare and tortoise move at the same speed
  int mu = 0;
  hare = x0;
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(hare);
    mu++;
  }
  // finding lambda, hare moves, tortoise stays
  int lambda = 1;
  hare = f(tortoise);
  while (tortoise != hare) {
    hare = f(hare);
    lambda++;
  }
  return ii(mu, lambda);
}
```

```
return factors; }        // if N does not fit in 32-bit integer and is a prime number
// then 'factors' will have to be changed to vector<ll>
// inside int main(), assuming sieve(1000000) has been called before
vi res = primeFactors(2147483647);                  // slowest, 2147483647 is a prime
res = primeFactors(136117223861LL);          // slow, 2 large pfactors 104729*1299709
res = primeFactors(142391208960LL);                   // faster, 2^10*3^4*5*7^4*11*13

// functions involving prime factors
// numPF(N):Count the number of prime factors of N
ll numPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF * PF <= N)) {
    while (N % PF == 0) { N /= PF; ans++; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}

// numDiffPF(N): count the number of different prime factors of N
ll numDiffPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    if (N % PF == 0) ans ++;                                // count this pf only once
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}

// sumPF(N): sum the prime factors of N
ll sumPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (PF * PF <= N) {
    while (N % PF == 0) { N /= PF; ans += PF; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans += N;
  return ans;
}

// numDiv(N): count the number of divisors of N
ll numDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;                 // start from ans = 1
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;                                                 // count the power
    while (N % PF == 0) { N /= PF; power++; }
    ans *= (power + 1);                                     // according to the formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= 2;                 // (last factor has pow = 1, we add 1 to it)
  return ans;
}
```

```
// sumDiv(N): sum the divisors of N
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;                  // start from ans = 1
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1)/(PF - 1);              // formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);            // last one
  return ans;
}

// EulerPhi(N): count the number of positive integers < N that are
// relatively prime to N.
ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;                  // start from ans = N
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans -= ans / PF;                     // only count unique factor
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans -= ans / N;                                          // last factor
  return ans;
}

// square matrix exponentiation
#define MAX_N 105                          // increase/decrease this value as needed
struct Matrix {
  int mat[MAX_N][MAX_N];                                // we will return a 2D array
};
Matrix matMul(Matrix a, Matrix b) {                                         // O(n^3)
  Matrix ans; int i, j, k;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      for (ans.mat[i][j] = k = 0; k < MAX_N; k++)                 // if necessary, use
        ans.mat[i][j] += a.mat[i][k] * b.mat[k][j];               // modulo arithmetic
  return ans;
}
Matrix matPow(Matrix base, int p) {                                   // O(n^3 log p)
  Matrix ans; int i, j;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      ans.mat[i][j] = (i == j);                              // prepare identity matrix
  while (p) {                 // iterative version of Divide & Conquer exponentiation
    if (p & 1) ans = matMul(ans, base);              // if p is odd (last bit is on)
    base = matMul(base, base);                                      // square the base
    p >>= 1;                                                          // divide p by 2
  }
  return ans;
}
```