Competitive Programming Team Notebook

BUBT_ModZero

Team Members:

Khairul Anam Mubin
Sharif Hossen
Shipul Roy

```
/***************************************BUBT_ModZero***********************************/
/*************************************Team Notebook************************************/
/******************String Hashing*******************/
// Double Hashing
// 1. Modular Exponentian Needed
// 2. Init must be call and set the maximum length of the string
// 3. If sub string hash required then Compute hash have to call
// 4. If prefix and suffix hash required ComputePreAndSufHash have to call
struct ModularExponentiation {
    template <typename T> T Pow(T b, T p) {
        T res = 1;
        while (p > 0) {
            if (p % 2 == 1) res = res * b;
            b = b * b;
            p /= 2;
        }
        return res;
    }
    template <typename T> T Mod(T a, T m) {
        return (((a % m) + m) % m);
    }
    template <typename T> T BigMod(T b, T p, T m) {
        T res = 1;
        if (b > m) b %= m;
        while (p) {
            if (p % 2 == 1) res = res * b % m;
            b = b * b % m;
            p /= 2;
        }
        return res;
    }
    template <typename T> T ModInv(T b,T m) {
        return BigMod(b , m - 2 , m);
    }
};
struct DoubleHashing {
    long long base[2] = {1949313259, 1997293877};
    long long mod[2] = {2091573227, 2117566807};
    vector <long long> pow[2] , inv[2];
    vector <long long> prehash[2] , sufhash[2];
    int maxN , flag = 0 , len;
    void Init(int n) {
        maxN = n + 2;
        for (int i = 0; i < 2; i++) {
            pow[i].resize(maxN);
            inv[i].resize(maxN);
        }
        Generate();
    }
    void Generate() {
        ModularExponentiation Ex;
        for (int j = 0; j < 2; j++) {
            pow[j][0] = 1;
            inv[j][0] = 1;
            long long minv = Ex.ModInv(base[j] ,mod[j]);
            for (int i = 1; i < maxN; i++) {
                pow[j][i] = pow[j][i - 1] * base[j] % mod[j];
                inv[j][i] = inv[j][i - 1] * minv % mod[j];
            }
        }
```

```
}
long long GetHash(string &s) {
    long long hash_val[2] = {0 , 0};
    int n = s.size();
    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < n; i++) {
            hash_val[j] = (hash_val[j] + s[i] * pow[j][i]) % mod[j];
        }
    }
    return (hash_val[0] << 32LL) | hash_val[1];
}
void ComputeHash(string &s) {
    flag = 1;
    len = s.size();
    for (int j = 0; j < 2; j++) prehash[j].resize(maxN);
    for (int j = 0; j < 2; j++) prehash[j][0] = 0;
    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < len; i++) {
            prehash[j][i + 1] = (prehash[j][i] + pow[j][i] * s[i]) % mod[j];
        }
    }
}
long long GetSubstrHash(int l , int r) {
    if (!flag) { cout << "ComputeHash\n"; return -1;}
    long long hash_val[2];
    for (int j = 0; j < 2; j++)
        hash_val[j] = (prehash[j][r + 1] - prehash[j][l]) * inv[j][l] % mod[j];
    for (int j = 0; j < 2; j++) if (hash_val[j] < 0) hash_val[j] += mod[j];
    return (hash_val[0] << 32) | hash_val[1];
}
void ComputePreAndSufHash(string &s) {
    flag = 1;
    len = s.size();
    for (int j = 0; j < 2; j++) {
        prehash[j].resize(maxN);
        sufhash[j].resize(maxN);
    }
    for (int j = 0; j < 2; j++) prehash[j][0] = sufhash[j][0] = 0;
    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < len; i++) {
            prehash[j][i + 1] = (prehash[j][i] + pow[j][i] * s[i]) % mod[j];
            sufhash[j][i + 1] = (sufhash[j][i] + pow[j][len - i + 1] * s[i]) % mod[j];
        }
    }
}
long long GetPrefixHash(int l , int r) {
    return GetSubstrHash(l , r);
}
long long GetSuffixHash(int l , int r) {
    if (!flag) { cout << "ComputePreAndSufHash\n"; return -1;}
    long long hash_val[2];
    for (int j = 0; j < 2; j++)
        hash_val[j] = (sufhash[j][r + 1] - sufhash[j][l]) * inv[j][len - r + 1] % mod[j];
    for (int j = 0; j < 2; j++) if (hash_val[j] < 0) hash_val[j] += mod[j];
    return (hash_val[0] << 32) | hash_val[1];
}
bool IsPallindrome(int l , int r) {
    return (GetPrefixHash(l , r) == GetSuffixHash(l , r));
}
vector <int> RabinKarp(string &txt , string &ptrn) {
        ComputeHash(txt);
        long long ptrn_hash = GetHash(ptrn);
        vector <int> occurences;
        int txtlen = txt.size();
        int ptrnlen = ptrn.size();
        for (int i = 0; i < txtlen - ptrnlen + 1; i++) {
            long long cur_hash = GetSubstrHash(i , ((i + ptrnlen) - 1));
            // pattern match...
            if (cur_hash == ptrn_hash)
                occurences.emplace_back(i + 1);
        }
        return occurences;
    }
} ;
/*******************************************************************************************/
/*****************************************KMP*********************************************/
// Building Prefix array
vector <int> BuildPrefixArray(string pattern) {
    vector <int> pfix(pattern.length()) ;
    pfix[0] = 0 ;
    for(int i = 1 ,j = 0 ; i < pattern.length() ; ) {
        if(pattern[i] == pattern[j])
            pfix[i++] = ++j ;
        else {
            if(j == 0) pfix[i++] = 0 ;
            else j = pfix[j - 1] ;
        }
    }
    return pfix ;
}
int Kmp(string text , string pattern) {
    vector <int> pfix = BuildPrefixArray(pattern) ;
    int cnt = 0 ;
    for(int i = 0 , j = 0 ; i < (int)text.length() && j < (int)pattern.length() ; ) {
        if(text[i] == pattern[j]) {
            i++ ;
            j++ ;
        }
        else {
            if(j == 0) i++ ;
            else j = pfix[j - 1] ;
        }
        if(j == (int)pattern.length()) {
            cnt++ ; // Number of occurances..
            j = pfix[j - 1] ;
        }
    }
    return cnt ;
}
```

Here we discuss two problems at once.

Given a string s of length n.

In the first variation of the problem we want to count the number of appearances of each prefix s[0…i] in the same string.

In the second variation of the problem another string t is given and we want to count the number of appearances of each prefix s[0…i] in t.

Let us compute the prefix function for s.
Using the last value of it we define the value k=n−π[n−1].
We will show, that if k divides n, then k will be the answer,

otherwise there doesn't exists an effective compression and the answer is n.

```
 vector<int> ans(n + 1);
for (int i = 0; i < n; i++)
    ans[pi[i]]++;
for (int i = n-1; i > 0; i--)
    ans[pi[i-1]] += ans[i];
for (int i = 0; i <= n; i++)
    ans[i]++;
/**********************************************************************************************/
/***************************************Suffix Array*******************************************/
template<typename T, bool maximum_mode = false>
struct RangeMinQuery {
    int n = 0;
    vector<vector<T>> sptab;
    static int LargestBit(int x) {
        return 31 - __builtin_clz(x);
    }
    static T Better(T a, T b) {
        return maximum_mode? max(a , b): min(a, b);
    }
    // O(NlogN)
    void Build(const vector<T> &values) {
        n = int(values.size());
        int levels = LargestBit(n) + 1;
        sptab.resize(levels);
        for (int k = 0; k < levels; k++) sptab[k].resize(n - (1 << k) + 1);
        if (n > 0) sptab[0] = values;
        for (int k = 1; k < levels; k++) {
            for (int i = 0; i <= n - (1 << k); i++) {
                sptab[k][i] = Better(sptab[k - 1][i], sptab[k - 1][i + (1 << (k - 1))]);
            }
        }
    }
    // O(1)
    int Query(int a, int b) const {
        assert(0 <= a && a < b && b <= n);
        int level = LargestBit(b - a);
        return Better(sptab[level][a], sptab[level][b - (1 << level)]);
    }
};
RangeMinQuery<int>rmq;
// Complexity O(N * LogN)
struct SuffixArray{
    string s;
    int n;
    vector <int> sar , lcp, rank; // suffix array, lcp , rank
    SuffixArray() {}
    SuffixArray(string _s) {
        Init(_s);
    }
    void Init(string _s) {
        s = _s;
        s += "$";
        n = s.size();
        sar.resize(n);
        rank.resize(n);
        lcp.resize(n);
        BuildSuffixArray();
        BuildLCPArray();
        rmq.Build(lcp);
    }
    // O(NLogN)
    void BuildSuffixArray() {
        vector <int> cnt(256, 0), pos(256);
        for (int i = 0; i < n; i++) cnt[s[i]]++;
        pos[0] = 0;
        for (int i = 1; i < 256; i++) pos[i] = pos[i - 1] + cnt[i - 1];
        for (int i = 0; i < n; i++) {
            sar[pos[s[i]]] = i;
            pos[s[i]]++;
        }
        rank[sar[0]] = 0;
        for (int i = 1; i < n; i++) {
            rank[sar[i]] = rank[sar[i - 1]];
            if (s[sar[i]] != s[sar[i - 1]]) rank[sar[i]]++;
        }
        pos.resize(n);
        for (int k = 0; (1 << k) < n; k++) {
            for (int i = 0; i < n; i++) sar[i] = (sar[i] - (1 << k) + n) % n;
            cnt.assign(n, 0);
            for (int x : rank) cnt[x]++;
            pos[0] = 0;
            for (int i = 1; i < n; i++) pos[i] = pos[i - 1] + cnt[i - 1];
            vector <int> sar_new(n);
            for (int x : sar) {
                int i = rank[x];
                sar_new[pos[i]] = x;
                pos[i]++;
            }
            sar = sar_new;
            vector <int> rank_new(n);
            rank_new[sar[0]] = 0;
            for (int i = 1; i < n; i++) {
                rank_new[sar[i]] = rank_new[sar[i - 1]];
                pair <int, int> prev = {rank[sar[i - 1]], rank[(sar[i - 1] + (1 << k)) %
n]};

                pair <int, int> cur = {rank[sar[i]] , rank[(sar[i] + (1 << k)) % n]};
                if (prev != cur) rank_new[sar[i]]++;
            }
            rank = rank_new;
        }
    }
    // algorithm of Kasai, Arimura, Arikawa, Lee and Park.
    // Complexity O(n)
    // Longest Common prefix of adjacent suffixes
    void BuildLCPArray() {
        int k = 0;
        for (int i = 0; i < n - 1; i++) {
            int pi = rank[i];
            int j = sar[pi - 1];
            while (s[i + k] == s[j + k]) k++;
            lcp[pi] = k;
            k = max(k - 1, 0);
        }
    }
    // O(1)
    int GetLcpFromRanks(int a, int b) const {
        if (a == b) return n - sar[a];
        if (a > b) swap(a, b);
        return rmq.Query(a + 1, b + 1);
    }
    // O(1)
    int GetLcp(int a, int b) const {
        if (a >= n || b >= n) return 0;
        if (a == b) return n - a;
        return GetLcpFromRanks(rank[a], rank[b]);
    }
    // Compares the substrings starting at `a` and `b` up to `length`
    // O(1)
    int Compare(int a, int b, int length = -1) const {
        if (length < 0) length = n;
        if (a == b) return 0;
        int common = GetLcp(a, b);
        if (common >= length) return 0;
        if (a + common >= n || b + common >= n) {
            return a + common >= n ? -1 : 1;
        }
        return s[a + common] < s[b + common] ? -1 : (s[a + common] == s[b + common] ? 0 :
1);
    }
    // O(N)
    long long NumOfUniqueSubStr() {
        long long sum = 0;
        for (int i = 1; i < lcp.size(); i++) sum += lcp[i];
        long long sz = n - 1; // actual size of string
        long long totalSubStrings = (sz * (sz + 1)) / 2LL;
        return totalSubStrings - sum;
    }
    // O(|t|*LogN)
    bool IsSubstr(string &t) {
        int tlen = t.size();
        if (tlen > n) return false;
        int low = LowerBound(t);
        if (low < n && n - sar[low] >= tlen && s.compare(sar[low] , tlen, t) == 0) return
true;
        return false;
    }
    // O(|t|*LogN)
    int LowerBound(string &t) {
        int low = 0 ,high = n ,tlen = t.size();
        while (low < high) {
            int mid = (low + high) >> 1;
            if (s.compare(sar[mid], tlen, t) < 0)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }
    // O(|t|*LogN)
    int UpperBound(string &t) {
        int low = 0 ,high = n ,tlen = t.size();
        while (low < high) {
            int mid = (low + high) >> 1;
            if (s.compare(sar[mid], tlen, t) <= 0)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }
    // O(|t|*LogN)
    int SubstrOccurences(string &t) {
        int tlen = t.size();
        int low = LowerBound(t) , up = UpperBound(t);
        if (low == up || s.compare(sar[low] , tlen , t) != 0) return 0;
        return (up - low);
    }
    // Longest common substring of two string
    // O(NLogN)
    string LongestCommonSubstring(string &a, string &b) {
        int alen = a.size();
        Init(a + "&" + b);
        int tot = sar.size();
        bool color[tot];
        for (int i = 0; i < tot; i++) {
            if (sar[i] < alen) color[i] = 1;
            else color[i] = 0;
        }
        int mx = -1 , mxid = -1;
        for (int i = 1; i < tot; i++) {
            if (color[i] != color[i - 1] && mx < lcp[i]) {
                mx = lcp[i];
                mxid = sar[i];
            }
        }
        if (mxid != -1) return s.substr(mxid , mx);
        return "";
    }
    void ShowSuffixArray() {
        cout << "Rank LCP    SA    Suffixes\n";
        for (int i = 0; i < n; i++) {
            cout << rank[i] << "    " << lcp[i] << "     " << sar[i] << "     " <<
s.substr(sar[i] , n - sar[i]) << "\n";
        }
    }
};
/***************************************************************************************/
/***********************************Trie***********************************************/
/*...........................Trie String matching............................*/
struct node{
    bool endmark ;
    node *next[27] ;
    node() {
        endmark = false ;
        for(int i = 0 ; i < 26 ; i++)
            next[i] = NULL ;
    }
};
node *root = new node() ;
void Insert(string str , int len) {
    node *curr = root ;
    for(int i = 0 ; i < len ; i++ ) {
        int id = str[i] - 'a' ;
        if(curr->next[id] == NULL)
            curr->next[id] = new node() ;
        curr = curr->next[id] ;
    }
    curr->endmark = true ;
```

```
}
bool Search(string str , int len) {
    node *curr = root ;
    for(int i = 0 ; i < len ; i++) {
        int id = str[i] - 'a' ;
        if(curr->next[id] == NULL)
            return false ;
        curr = curr->next[id] ;
    }
    return curr->endmark ;
}

/**************************************************************************************/
/************************Manacher's Algorithm*****************************/
/*
    Algorithm:
    1. Finds all sub-palindromes in O(N) , Applications: 1. Finds Longest Palindromes O(N)
*/
/*
    1. string is 0 based indexed of length n.
    2. the manacher will make a string of length 2 * n + 1.
        ex: "abba" will be "#a#b#b#a#".
    3. Odd length palindromes:
        for all i = 0 to i < n
            i is a center and stored maxlen palindrome in 2i + 1 th index in P.
    4. Even length palindromes:
        there are n - 1 centered positions.
        for all i = 0 to i < n - 1
            max length palindrome centered at i found in 2i + 2 th index in P.
*/
vector<int> Manacher_Odd_Length(string T) {
    int n = T.size();
    vector <int> P(n);
    int C = 0, R = -1, rad;
    for (int i = 0; i < n; ++i) {
        if (i <= R) {
            rad = min(P[2 * C - i], R - i);
        } else {
            rad = 0;
        }
        // Try to extend
        while (i + rad < n && i - rad >= 0 && T[i - rad] == T[i + rad]) {
            rad++;
        }
        P[i] = rad;
        if (i + rad - 1 > R) {
            C = i;
            R = i + rad - 1;
        }
    }
    for (int i = 0; i < (int)P.size(); i ++) {
        P[i]--;
    }
    return P;
}
vector <int> Manacher(const string &s) {
    string t;
    for(auto c: s) {
        t += string("#") + c;
    }
    return Manacher_Odd_Length(t + "#");
}
string LongestPalindrome(const string &s) {
    vector <int> P = Manacher(s);
    int mxlen = 0, startpos = -1;
    for (int i = 0; i < (int)s.size(); i++) {
        if (mxlen < P[2 * i + 1]) {
            mxlen = P[2 * i + 1];
            startpos = i - P[2 * i + 1] / 2;
        }
    }
    for (int i = 0; i < (int)s.size() - 1; i++) {
        if (mxlen < P[2 * i + 2]) {
            mxlen = P[2 * i + 2];
            startpos = i - P[2 * i + 2] / 2 + 1;
        }
    }
    return s.substr(startpos, mxlen);
}
/***********************************Data Structure************************************/
/************************************Segment tree*************************************/
/************************Segment tree**********************/

#define INF9         2147483647
#define INF18        9223372036854775806
template <typename T> struct SegmentTree {
    vector <T> seg;
    vector <T> lazy;
    vector <T> ar;
    int type , up;
    SegmentTree() {
        type = 0;
        up = 0;
    }
    SegmentTree(int tp , int u) {
        type = tp;
        up = u;
    }
    void Init(int N) {
        seg.assign(N << 2, 0);
        lazy.assign(N << 2 , 0);
    }
    void Init(vector <T> &s) {
        Init(s.size() + 1);
        ar = s;
    }
    void PushDown(int cur , int left , int right) {
        if (type == 0) {
            if (up == 1) seg[cur] += (right - left + 1) * lazy[cur];
            else seg[cur] = (right - left + 1) * lazy[cur];
        } else {
            if (up == 1) seg[cur] += lazy[cur];
            else seg[cur] = lazy[cur];
        }
        if (left != right) {
            if (up == 1) {
                lazy[cur << 1] += lazy[cur];
                lazy[cur << 1 | 1] += lazy[cur];
            } else {
                lazy[cur << 1] = lazy[cur];
                lazy[cur << 1 | 1] = lazy[cur];
            }
        }
        lazy[cur] = 0;
    }
    T Merge(T x , T y) {
        if (type == 0) return x + y;
        if (type == 1) return max(x , y);
        if (type == 2) return min(x , y);
    }
    void Build(int cur , int left , int right) {
        lazy[cur] = 0;
        if (left == right) {
            seg[cur] = ar[left];
            return;
        }
        int mid = (left + right) >> 1;
        Build(cur << 1 , left , mid);
        Build(cur << 1 | 1 , mid + 1 , right);
        seg[cur] = Merge(seg[cur << 1] , seg[cur << 1 | 1]);
    }
    void Update(int cur , int left , int right , int pos , T val) {
        Update(cur , left , right , pos , pos , val);
    }
    void Update(int cur , int left , int right , int l , int r , T val) {
        if (lazy[cur] != 0) PushDown(cur , left , right);
        if (l > right || r < left) return;
        if (left >= l && right <= r) {
            if (up == 0) lazy[cur] = val;
            else lazy[cur] += val;
            PushDown(cur , left , right);
            return ;
        }
        int mid = (left + right) >> 1;
        Update(cur << 1 , left , mid , l , r , val);
        Update(cur << 1 | 1 , mid + 1 , right , l , r , val);
        seg[cur] = Merge(seg[cur << 1] , seg[cur << 1 | 1]);
    }
    T Query(int cur , int left , int right , int l , int r) {
        if (l > right || r < left) {
            if (type == 0) return 0;
            if (type == 1) return -INF18;
            if (type == 2) return INF18;
        }
        if (lazy[cur] != 0) PushDown(cur , left , right);
        if (left >= l && right <= r) return seg[cur];
        int mid = (left + right) >> 1;
        T p1 = Query(cur << 1 , left , mid , l , r);
        T p2 = Query(cur << 1 | 1 , mid + 1 , right , l , r);
        return Merge(p1 , p2);
    }
};
//for sum = 0, max = 1, min = 2, for assignment update send 0 or 1 for increment.
SegmentTree <long long> tr(0 , 0);

/****************2D Segment tree***********************/
const int mxN = 1e3;
int a[mxN + 1][mxN + 1];
int t[mxN << 2][mxN << 2];
int m;
```

```
void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
    if (ly == ry) {
        if (lx == rx)
            t[vx][vy] = a[lx][ly];
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    } else {
        int my = (ly + ry) / 2;
        build_y(vx, lx, rx, vy*2, ly, my);
        build_y(vx, lx, rx, vy*2+1, my+1, ry);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}
void build_x(int vx, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        build_x(vx*2, lx, mx);
        build_x(vx*2+1, mx+1, rx);
    }
    build_y(vx, lx, rx, 1, 0, m-1);
}
int sum_y(int vx, int vy, int tly, int try_, int ly, int ry) {
    if (ly > ry)
        return 0;
    if (ly == tly && try_ == ry)
        return t[vx][vy];
    int tmy = (tly + try_) / 2;
    return sum_y(vx, vy*2, tly, tmy, ly, min(ry, tmy))
         + sum_y(vx, vy*2+1, tmy+1, try_, max(ly, tmy+1), ry);
}
int sum_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
    if (lx > rx)
        return 0;
    if (lx == tlx && trx == rx)
        return sum_y(vx, 1, 0, m-1, ly, ry);
    int tmx = (tlx + trx) / 2;
    return sum_x(vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)
         + sum_x(vx*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry);
}
void update_y(int vx, int lx, int rx, int vy, int ly, int ry, int x, int y, int new_val) {
    if (ly == ry) {
        if (lx == rx)
            t[vx][vy] ^= 1;
        else
            t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
    } else {
        int my = (ly + ry) / 2;
        if (y <= my)
            update_y(vx, lx, rx, vy*2, ly, my, x, y, new_val);
        else
            update_y(vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
    }
}
void update_x(int vx, int lx, int rx, int x, int y, int new_val) {
    if (lx != rx) {
        int mx = (lx + rx) / 2;
        if (x <= mx)
            update_x(vx*2, lx, mx, x, y, new_val);
        else
            update_x(vx*2+1, mx+1, rx, x, y, new_val);
    }
    update_y(vx, lx, rx, 1, 0, m-1, x, y, new_val);
}
int main() {
    int n, q; cin >> n >> q;
    m = n;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            char ch; cin >> ch;
            a[i][j] = (ch == '*')? 1 : 0;
        }
    }
    build_x(1 , 0 , m - 1);
    while (q--) {
        int typ; cin >> typ;
        if (typ == 2) {
            int y1, x1, y2, x2;
            cin >> y1 >> x1 >> y2 >> x2;
            y1--, x1--,y2--,x2--;
            cout << sum_x(1 , 0 , m - 1, y1 , y2 , x1 , x2) << '\n';
        } else {
            int y, x; cin >> y >> x;
            y--,x--;
            update_x(1 , 0 , m - 1, y , x , 1);
        }
    }
    return  0;
}/*********************************************************************************************/
/*********************************Disjoint Set Union Find*******************************/ //
// 1. assign max size with constructor
// 2. each test case call makeset
struct DSU {
    vector <int> parent;
    vector <int> siz;
    DSU(int mxN) {
        mxN <<= 1;
        parent.resize(mxN + 1);
        siz.resize(mxN + 1);
    }
    void Makeset(int n) {
        for (int i = 1 ; i <= n ; i++) {
            parent[i] = n + i;
            parent[n + i] = n + i;
            siz[n + i] = 1;
        }
    }
    int Find(int u) {
        if (parent[u] == u) return u ;
        return parent[u] = Find(parent[u]) ;
    }
    void Union(int u , int v) {
        u = Find(u);
        v = Find(v);
        if (u != v) {
            if (siz[u] < siz[v]) swap(u , v);
            parent[v] = u;
            siz[u] += siz[v];
        }
    }
    bool SameSet(int u , int v) {
        return (Find(u) == Find(v)) ;
    }
    void MoveUtoSetV(int u , int v) {
        if (SameSet(u , v)) return;
        int x = Find(u);
        int y = Find(v);
        siz[x]--;
        siz[y]++;
        parent[u] = y;
    }
    int Size(int u) {
        return siz[Find(u)];
    }
};
/*******************************************************************************************/
/*************************************HeavyLightDecomposition************************* *****/
/*  1.  All nodes are number from 0 to n - 1  */
/*  2.  Assign the graph by Init(graph) or simply Init(total nodes) and
        call AddEdge(u , v) for all the edges */
/*  3.  Must be Take the node value from input directly or use the
        TakeNodeVal(nodeval) to assigning the node value */
/*  4.  Call Build() to construct hld and segment tree */
/*  5.  simply use the path query by query(u , v) and update(pos , val)
/*
/*  6. use optimized segment tree sometimes it cz TLE
/**********************Segment tree***********************************/
//for sum = 0, max = 1, min = 2, for assignment update send 0 or 1 for increment.
SegmentTree <long long> T(0 , 0);
struct HeavyLightDecompose {
    vector <vector <int> > g ; // graph
    vector <long long> node_val;
    int N , root = 0;
    vector <int> depth , parent , sub;
    // HLD staffs
    int chain_no, indx;
    vector <int> chain_head , chain_ind;
    vector <int> node_serial , serial_node;
    vector <long long> segarr; // tree on linear format
    void Init(int n) {
        N = n;
        g.assign(N , {});
        node_val.clear();
        segarr.resize(N);
        depth.resize(N);
        parent.resize(N);
        sub.resize(N);
        chain_head.assign(N, -1);
        chain_ind.resize(N);
        node_serial.resize(N);
        serial_node.resize(N);
        return;
    }
    void Init(const vector <vector<int>> &_g) {
        Init(_g.size());
        g = _g;
        return;
    }
    void AddEdge(int u , int v) {
```

```
    g[u].push_back(v) ;
    g[v].push_back(u) ;
    return;
}
void TakeNodeVal(const vector <long long> &_node_val) {
    node_val = _node_val;
}
void Build() {
    Dfs(root);
    chain_no = 0, indx = 0;
    HLD(0);
    T.Init(segarr);
    T.Build(1 , 0 , N - 1);
}
void Dfs(int u, int par = -1) {
    sub[u] = 1;
    if (par == -1) {
        depth[u] = 0;
        parent[u] = -1;
    }
    for (int v : g[u]) {
        if (v == par) continue;
        parent[v] = u;
        depth[v] = depth[u] + 1;
        Dfs(v , u);
        sub[u] += sub[v];

    }
    return;
}
void HLD(int u , int par = -1) {
    if (chain_head[chain_no] == -1) chain_head[chain_no] = u;
    chain_ind[u] = chain_no;
    node_serial[u] = indx;
    serial_node[indx] = u;
    segarr[indx] = node_val[u]; // tree flatting..
    indx++;
    int heavychild = -1 , heavysize = 0;
    for (int v : g[u]) {
        if (v == par) continue;
        if (sub[v] > heavysize) {
            heavysize = sub[v];
            heavychild = v;
        }
    }
    if (heavychild != -1) HLD(heavychild , u);
    for (int v : g[u]) {
        if (v != par && v != heavychild) {
            chain_no++;
            HLD(v , u);
        }
    }
    return;
}
void Update(int p , int val) {
    T.Update(1 , 0 , N - 1, node_serial[p] , val);
    node_val[p] = val;
}
long long Query(int u , int v) {
    long long ans = 0;
```

```
        for ( ; chain_ind[u] != chain_ind[v] ; v = parent[chain_head[chain_ind[v]]]) {
            if (depth[chain_head[chain_ind[u]]] > depth[chain_head[chain_ind[v]]])
                swap( u , v );
            ans += T.Query(1 , 0 , N - 1 , node_serial[chain_head[chain_ind[v]]] ,
node_serial[v]);
        }
        if (depth[u] > depth[v])
            swap(u , v);
        ans += T.Query(1 , 0 , N - 1 , node_serial[u] , node_serial[v]);
        return ans;
    }
} hd;
/********************************************************************************************/
/****************************************************************************************/
/**************************************MO's ALgorithm************************************/
/*.............Range query..............*/
/* Given an array of length n and q querys of range l , r . Find the number of unique
elements in the given range */

const int BLOCK = 555;
const int mxN = 100000;
struct query {
    int l , r , idx;
};
query Q[mxN + 5];
int ar[mxN + 5] , ans[mxN + 5];
int freq[mxN + 5];
int cnt = 0;
bool Cmp(query &a , query &b) {
    if (a.l / BLOCK != b.l / BLOCK)
        return a.l / BLOCK < b.l / BLOCK;
    return a.r < b.r;
}
void Add(int pos) {
    freq[ar[pos]]++;
    if (freq[ar[pos]] == 1) cnt++;
}
void Remove(int pos) {
    freq[ar[pos]]--;
    if (freq[ar[pos]] == 0) cnt--;
}
void Input_Query(int q) {
    for (int i = 0 ; i < q ; i++) {
        cin >> Q[i].l >> Q[i].r;
        Q[i].idx = i;
        Q[i].l--; Q[i].r--;
    }
}
void MosAlgo(int q) {
    Input_Query(q);
    sort(Q , Q + q , Cmp);
    int ML = 0 , MR = -1;
    for (int i = 0 ; i < q ; i++) {
        int L = Q[i].l;
        int R = Q[i].r;
        while (ML > L) Add(--ML);
        while (MR < R) Add(++MR);
        while (ML < L) Remove(ML++);
        while (MR > R) Remove(MR--);
        ans[Q[i].idx] = cnt;
    }
}
int main() {
    FasterIO
    int n , q ;
    cin >> n ;
    for(int i = 0 ; i < n ; i++)
        cin >> ar[i] ;
    cin >> q ;
    MosAlgo(q) ;
    for(int i = 0 ; i < q ; i++) {
        cout << ans[i] << "\n" ;
    }
    return 0 ;
}/****************************************************************************************/
/*****************************************Wavelet Tree***********************************/
/***********************Wavelet Tree**************************/
// 1 based index...
const int N = 3e5, M = N;
const int MAX = 1e6;
int a[N];
struct wavelet_tree {
    int lo, hi;
    wavelet_tree *l, *r;
    vector <int> b;
    vector <int> c; // c holds the prefix sum of elements
    //nos are in range [x,y]
    //array indices are [from, to)
    wavelet_tree(int *from, int *to, int x, int y) {
        lo = x, hi = y;
        if( from >= to) return;
        if( hi == lo ) {
            b.reserve(to - from + 1);
            b.push_back(0);
            c.reserve(to - from + 1);
            c.push_back(0);
            for(auto it = from; it != to; it++) {
                b.push_back(b.back() + 1);
                c.push_back(c.back() + *it);
            }
            return ;
        }
        int mid = (lo + hi) / 2;
        auto f = [mid](int x) {
            return x <= mid;
        };
        b.reserve(to - from + 1);
        b.push_back(0);
        c.reserve(to - from + 1);
        c.push_back(0);
        for(auto it = from; it != to; it++) {
            b.push_back(b.back() + f(*it));
            c.push_back(c.back() + *it);
        }
        //stable_partition the lamda function
        auto pivot = stable_partition(from, to, f);
        l = new wavelet_tree(from, pivot, lo, mid);
        r = new wavelet_tree(pivot, to, mid + 1, hi);
    }
    // swap a[i] with a[i+1]  , if a[i]!=a[i+1] call swapadjacent(i)
    void swapadjacent(int i) {
        if(lo == hi) return ;
        b[i] = b[i - 1] + b[i + 1] - b[i];
        c[i] = c[i - 1] + c[i + 1] - c[i];
        if( b[i + 1] - b[i] == b[i] - b[i - 1]) {
            if(b[i] - b[i - 1])
                return this->l->swapadjacent(b[i]);
            else
                return this->r->swapadjacent(i - b[i]);
        }
        return ;
    }
    //kth smallest element in [l, r]
    int kth(int l, int r, int k) {
        if(l > r) return 0;
        if(lo == hi) return lo;
        int inLeft = b[r] - b[l - 1];
        int lb = b[l - 1]; //amt of nos in first (l-1) nos that go in left
        int rb = b[r]; //amt of nos in first (r) nos that go in left
        if(k <= inLeft) return this->l->kth(lb + 1, rb, k);
        return this->r->kth(l - lb, r - rb, k - inLeft);
    }
    //count of nos in [l, r] Less than or equal to k
    int LTE(int l, int r, int k) {
        if(l > r || k < lo) return 0;
        if(hi <= k) return r - l + 1;
        int lb = b[l-1], rb = b[r];
        return this->l->LTE(lb + 1, rb, k) + this->r->LTE(l - lb, r - rb, k);
    }
    //count of nos in [l, r] equal to k
    int count(int l, int r, int k) {
        if(l > r || k < lo || k > hi) return 0;
        if(lo == hi) return r - l + 1;
        int lb = b[l - 1], rb = b[r], mid = (lo + hi) / 2;
        if(k <= mid) return this->l->count(lb + 1, rb, k);
        return this->r->count(l - lb, r - rb, k);
    }
    //sum of nos in [l ,r] less than or equal to k
    int sumk(int l, int r, int k) {
        if(l > r || k < lo) return 0;
        if(hi <= k) return c[r] - c[l-1];
        int lb = b[l - 1], rb = b[r];
        return this->l->sumk(lb + 1, rb, k) + this->r->sumk(l - lb, r - rb, k);
    }
    ~wavelet_tree() {
        delete l;
        delete r;
    }
};
int main() {
    int n ; cin >> n;
    for(int i = 1 ; i <= n ; i++) {
        cin >> a[i] ;
    }
    // wavelet_tree T(array start address, array end address, min element, max element) ;
    wavelet_tree T(a + 1, a + n + 1 , 1, MAX) ;
    int q ; cin >> q ;
    while(q--) {
        int x ; cin >> x;
        int k , l , r ;
        if(x == 0) {
            //kth smallest element in range [l , r]
            cin >> l >> r >> k;
            cout << "Kth smallest: ";
            cout << T.kth(l, r, k) << endl;
        }
        if(x == 1) {
            //Number of Elements less than or equal to K in range [l , r]
            cin >> l >> r >> k;
            cout << "LTE: ";
            cout << T.LTE(l, r, k) << endl;
        }
        if(x == 2) {
            //count occurence of K in [l, r]
            cin >> l >> r >> k;
            cout << "Occurence of K: ";
            cout << T.count(l, r, k) << endl;
        }
        if(x == 3) {
            //sum of elements less than or equal to K in [l, r]
            cin >> l >> r >> k;
            cout << "Sum: ";
            cout << T.sumk(l, r, k) << endl;
        }
        if(x == 4) {
            int pos ; cin >> pos ;
            if(a[pos] != a[pos + 1])
                T.swapadjacent(pos) ;
        }
    }
    return 0;
}
/********************************************************************************************/
/***********************Lowest Common Ancestor***********************/
/* 1.   All nodes are number 0 to n - 1 */
/* 2.   simply Init(total nodes) and call AddEdge(u , v) for all the edges */
/* 3.   Call Build() to run dfs and build the sparse table */

struct LowestCommonAncestor {
    int N , root = 0, po;

    vector <vector <int> > g;
    vector <vector <int> > sptab;
    vector <int> depth;
    vector <int> parent;

    void Init(int _n) {
        N = _n;
        po = log2((N)) + 1;
        g.assign(N, {});
        depth.resize(N);
        parent.resize(N);
        sptab.assign(N, {});
    }
    void AddEdge(int u , int v) {
        g[u].push_back(v);
        g[v].push_back(u);
    }
    void Dfs(int u , int par = -1) {
        if(par == -1) {
```

```
        depth[u] = 0;
        parent[u] = -1;
    }
    for(int v : g[u]) {
        if (v == par) continue;
        parent[v] = u;
        depth[v] = depth[u] + 1;
        Dfs(v , u);
    }
}
void SparceTable() {
    for(int i = 0 ; i < N ; i++) sptab[i][0] = parent[i];
    for(int j = 1 ; (1 << j) < N ; j++) {
        for(int i = 0 ; i < N ; i++) {
            if(sptab[i][j - 1] != -1) {
                sptab[i][j] = sptab[sptab[i][j - 1]][j - 1];
            }
        }
    }
}
void Build() {
    for(int i = 0 ; i < N ; i++) {
        for(int j = 0 ; j <= po ; j++) {
            sptab[i].push_back(-1);
        }
    }
    Dfs(root);
    SparceTable();
}
int Lca(int u , int v) {
    if(depth[u] < depth[v]) swap(u , v);
    int log;
    for(log = 1 ; (1 << log) <= depth[u] ; log++); log--;
    for(int i = log ; i >= 0 ; i--) {
        if(depth[u] - (1 << i) >= depth[v]) {
            u = sptab[u][i];
        }
    }
    if(u == v) return u;
    for(int i = log ; i >= 0 ; i--) {
        if(sptab[u][i] != -1 && sptab[u][i] != sptab[v][i]) {
            u = sptab[u][i];
            v = sptab[v][i];
        }
    }
    return parent[u];
}
int KthAncestor(int u, int k) {
    int log;
    for(log = 1 ; (1 << log) <= depth[u] ; log++); log--;
    for(int i = log ; i >= 0 ; i--) {
        if(k - (1 << i) >= 0) {
            u = sptab[u][i];
            k -= (1 << i);
        }
    }
    return u;
}
int Getdist(int u , int v) {
    return (depth[u] + depth[v] - (2 * (depth[Lca(u , v)]))) ;
    }
    bool IsAnsector(int u , int v) {
        int cur = Lca(u , v);
        if(cur == u) return 1;
        return 0;
    }
} lca;
/*******************************Graph************************************************/
/*********************************SSC and Topsort*********************************************/
struct StronglyConnectedComponent {
    vector <vector <int>> g , gr;
    vector <bool> vis;
    vector <int> order, sccid;
    int nodes , edges , scc = 0;

    void Init(int _nodes) {
        nodes = _nodes;
        edges = 0;
        scc = 0;
        g.clear();
        gr.clear();
        g.assign(nodes + 1 , {});
        gr.assign(nodes + 1, {});
        order.clear();
        vis.assign(nodes + 1, 0);
        sccid.resize(nodes + 1);
    }
    void AddEdge(int u , int v) {
        if (u == -1 || v == -1) return;
        g[u].push_back(v);
        gr[v].push_back(u);
        edges++;
    }
    void Init() {
        for (int i = 0; i <= nodes; i++) vis[i] = 0;
    }
    void Dfs1(int u) {
        if (vis[u]) return;
        vis[u] = 1;
        for (int v : g[u]) Dfs1(v);
        order.push_back(u);
    }
    void TopSort() {
        Init();
        for (int i = 0; i < nodes; i++) {
            if (!vis[i]) Dfs1(i);
        }
        reverse(order.begin() , order.end());
    }
    void Dfs2(int u, int id) {
        if (vis[u]) return;
        vis[u] = 1;
        sccid[u] = id;
        for (int v : gr[u]) Dfs2(v , id);
    }
    void KosarajuSCC() {
        TopSort();
        Init();
        for (int i : order) {
            if (!vis[i]) {
                Dfs2(i , scc);
                scc++;
            }
        }
    }
    int GetAns() {
        vector <int> here(nodes + 1);
        for (int i = 0; i < scc; i++ ) {
            here[i] = 1;
        }
        for (int i = 0; i < nodes; i++) {
            for (int j : g[i]) {
                if (sccid[i] != sccid[j])
                    here[sccid[j]] = 0;
            }
        }
        int ans = 0;
        for (int i = 0; i < scc; i++) ans += here[i];
        return ans;
    }
};
/*********************************Strongly ConnectComponent(Kosaraju_dfs)********************/

/*...........................Finding strongly connected component............................*/
/* Given a graph , Find if there is one or more connected component. print 1 for 1 or 0 for
more.. */
// complexity : O(V + E)
#define mxN 2000
vector <int> G[mxN + 10] , GR[mxN + 10] ;
vector <int> order , component ;
bool vis[mxN + 10] ;

void init(int nodes) {
    for(int i = 0 ; i <= nodes ; i++)
        vis[i] = 0 ;
}
void dfs1(int u) {
    vis[u] = 1 ;
    for(int v : G[u]) {
        if(!vis[v])
            dfs1(v) ;
    }
    order.push_back(u) ;
}
void reverse_edges(int nodes , int edges) {
    for(int i = 1 ; i <= nodes ; i++) {
        for(int j : G[i]) {
            GR[j].push_back(i) ;
        }
    }
}
void dfs2(int u) {
    vis[u] = 1 ;
    component.push_back(u) ;
    for(int v : GR[u]) {
        if(!vis[v])
            dfs2(v) ;
    }
}
void KosarajuSCC(int nodes ,int edges) {
    init(nodes) ;
    for(int i = 1 ; i <= nodes ; i++) {
        if(!vis[i])
            dfs1(i) ;
    }
    reverse_edges(nodes , edges) ;
    init(nodes) ;
    int cnt = 0 ;
    for(int i = order.size() - 1 ; i >= 0 ; i--) {
        int v = order[i] ;
        if(!vis[v]) {
            dfs2(v) ;
            cnt++ ; // Component containg here all the nodes which are in a SCC..
            component.clear() ;
        }
    }
    if(cnt == 1)
        cout << "1\n" ;
    else
        cout << "0\n" ;
}
int main() {
    int nodes , edges ;
    while(cin >> nodes >> edges) {
        if(nodes == edges && nodes == 0)
            break ;
        for(int i = 0 ; i < edges ; i++) {
            int u , v , w ;
            cin >> u >> v >> w ;
            if(w == 1) {
                G[u].push_back(v) ;
            } else {
                G[u].push_back(v) ;
                G[v].push_back(u) ;
            }
        }
        KosarajuSCC(nodes ,  edges) ;
        graph_clear(nodes) ;
    }
    return 0 ;
}/******************************************************************************************/
/*****************************Shortest Path Faster Algorithm******************************/
#define INF 1e9
vector < pair <int , int> > G[1010] ;
int cost[1010] ; // containing shortest path costs..
int vis[1010] ;

// The chinese algorithm..
void ShortestPathFasterAlgorithm(int nodes) {
    for(int i = 0 ; i < 1010 ; i++) {
        cost[i] = INF ;
        vis[i] = 0 ;
    }
    cost[0] = 0 ; // setting the source cost as 0
    queue <int> q ;
    q.push(0) ;
    while(!q.empty()) {
        int u = q.front() ;
        q.pop() ;
        vis[u] = 1 ;
        for(int i = 0 ; i < G[u].size() ; i++) {
            int v = G[u][i].first ;
            int wv = G[u][i].second ;
            if(cost[u] + wv < cost[v]) {
                cost[v] = cost[u] + wv ;
                if(!vis[v]) {
                    q.push(v) ;
                    vis[v] = 1 ;
                }
            }
        }
    }
}
// If there exist a negative weight cycle returning true
bool NegativeCycle(int nodes) {
    for(int u = 0 ; u < nodes ; u++) {
        for(int i = 0 ; i < G[u].size() ; i++) {
            int v = G[u][i].first ;
            int wv = G[u][i].second ;
            if(cost[v] > cost[u] + wv)
                return 1 ;
        }
    }
    return 0 ;
}
/********************************************************************************************/
/*****************************************Number Theory***********************************/
/******************************Big Prime and Factorisation****************************/
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
/*
支持更高的快速幂操作
expower.mod_pow(a,b,mod);
*/
struct Expower {
    ull Mod_mul(ull a, ull b, ull M) {
        ll ret = a * b - M * ull(ld(a) * ld(b) / ld(M));
        return ret + M * (ret < 0) - M * (ret >= (ll)M);
    }
    ull Mod_pow(ull b, ull e, ull mod) {
        ull ans = 1;
        for ( ; e ; b = Mod_mul( b, b, mod), e /= 2)
            if (e & 1) ans = Mod_mul(ans, b, mod);
        return ans % mod;
    }
} Expower;
struct BigPrime {
    /*
    Miller-Rubin 素数判别
    is_prime(n);
    */
    bool IsPrime(ull n) {
        if (n < 2 || n % 6 % 4 != 1)
            return n - 2 < 2;
        ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
        ull s = __builtin_ctzll(n - 1), d = n >> s;
        for (auto a : A) {
            ull p = Expower.Mod_pow(a, d, n), i = s;
            while (p != 1 && p != n - 1 && a % n && i--)
                p = Expower.Mod_mul(p, p, n);
            if (p != n - 1 && i != s)
                return 0;
        }
        return 1;
    }
    /*
    素因数分解
    ret=factorization(n);
    */
    ull Pollard(ull n) {
        auto f = [n](ull x) {
            return ( Expower.Mod_mul(x, x, n) + 1) % n;
        };
        if (!( n & 1))
            return 2;
        for (ull i = 2 ; ; i++) {
            ull x = i, y = f(x), p;
            while ((p = __gcd( n + y - x , n)) == 1)
                x = f(x) , y = f(f(y));
            if (p != n) return p;
        }
    }
    vector <ull> Factorization(ull n){
        if (n == 1) return {};
        if (IsPrime(n)) return {n};
        ull x = Pollard(n);
        auto l = Factorization(x), r = Factorization(n/x);
        l.insert(l.end(), begin(r), end(r));
        return l;
    }
};
int main() {
    FasterIO
    BigPrime ob;
    int tc; cin >> tc;
    while (tc--) {
        ull n ; cin >> n;
        if (ob.IsPrime(n)) {
            cout << "YES\n";
        } else {
            cout << "NO\n";
        }
    }
    return 0 ;
}
/***************************************************************************************/
/*****************************************Phi*********************************************
/ // Build Complexity : O(NlogN)
void SieveOfEulersPhi() {
    for (int i = 0; i <= mxN; i++) phi[i] = 0;
    phi[1] = 1;
    for (int i = 2; i <= mxN; i++) {
        if (phi[i] == 0) {
            phi[i] = i - 1;
            for (int j = i + i; j <= mxN; j += i) {
                if (phi[j] == 0) phi[j] = j;
                phi[j] -= phi[j] / i;
            }
        }
    }
}
// if n <= 1e7, always works with O(1) per query
// if n >= 1e7, most often works with O(logN) per query but sometimes goes O(sqrtN) in very
rare.
long long Phi(long long n) {
    if (n <= mxN) return phi[n];
    long long coprime = n;
    for (int i = 0; i < tot_primes; i++) {
        if (n <= mxN && isp[n] == 0) break;
        long long x = prime[i];
        if (x * x > n) break;
        if (n % x == 0) {
            while (n % x == 0) n /= x;
            coprime -= coprime / x;
        }
    }
    if (n != 1) coprime -= coprime / n;
    return coprime;
}
/*************************************************************************************/
// Sum of Number of divisors in range 1 to N .
// Complexity O(sqrt(N))
int SNOD( int n ) {
int res = 0;     int u =
sqrt(n);
    for ( int i = 1; i <= u; i++ ) {
res += ( n / i ) - i; //Step 1
    }
    res *= 2; //Step 2
res += u; //Step 3
return res;
}

// sum of coprimes of n int
sumofcoprimesN(int n){
int x = phi(n);     int ans
= (x * n) / 2 ;     return
ans;
}
/******************************************************************************************/
// Sum of Number of divisors in range 1 to N .
// Complexity O(sqrt(N))
int SNOD( int n ) {
    int res = 0;
    int u = sqrt(n);
    for ( int i = 1; i <= u; i++ ) {
        res += ( n / i ) - i; //Step 1
    }
    res *= 2; //Step 2
    res += u; //Step 3
    return res;
}
// sum of coprimes of n
int sumofcoprimesN(int n){
    int x = phi(n);
    int ans = (x * n) / 2 ;
    return ans;
}
```