```cpp
// Aho-Corasick
// Complexity : O(n+m+z)
// n : Length of text
// m : total length of all keywords
// z : total number of occurance of word in text

const int TOTKEY = 505;                   // Total number of keywords
const int KEYLEN = 505;                   // Size of maximum keyword
const int MAXS = TOTKEY*KEYLEN + 10;      // Max number of states in the matching machine.
                                          // Should be equal to the sum of the length of all keywords.
const int MAXC = 26;                      // Number of characters in the alphabet.
bitset<TOTKEY> out[MAXS];                 // Output for each state, as a bitwise mask.
int f[MAXS];                              // Failure function
int g[MAXS][MAXC];                        // Goto function, or -1 if fail.

int buildMatchingMachine(const vector<string> &words, char lowestChar = 'a', char highestChar = 'z') {
    for(int i = 0; i < MAXS; ++i)
        out[i].reset();
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);

    int states = 1;                                         // Initially, we just have the 0 state
    for(int i = 0; i < (int)words.size(); ++i) {
        const string &keyword = words[i];
        int currentState = 0;
        for(int j = 0; j < (int)keyword.size(); ++j) {
            int c = keyword[j] - lowestChar;
            if(g[currentState][c] == -1)                    // Allocate a new node
                g[currentState][c] = states++;
            currentState = g[currentState][c];
        }
        out[currentState].set(i);                           // There's a match of keywords[i] at node currentState.
    }

    for(int c = 0; c < MAXC; ++c)                           // State 0 should have an outgoing edge for all characters.
        if(g[0][c] == -1)
            g[0][c] = 0;
                                                                // Now, let's build the failure function
    queue<int> q;
    for(int c = 0; c <= highestChar - lowestChar; ++c)          // Iterate over every possible input
        if(g[0][c] != -1 and g[0][c] != 0) {                    // All nodes s of depth 1 have f[s] = 0
            f[g[0][c]] = 0;
            q.push(g[0][c]);
        }

    while(q.size()) {
        int state = q.front();
        q.pop();
        for(int c = 0; c <= highestChar - lowestChar; ++c) {
            if(g[state][c] != -1) {
                int failure = f[state];

                while(g[failure][c] == -1)
                    failure = f[failure];

                failure = g[failure][c];
                f[g[state][c]] = failure;
                out[g[state][c]] |= out[failure];               // Merge out values
                q.push(g[state][c]);
            } } }
    return states;
}

int findNextState(int currentState, char nextInput, char lowestChar = 'a') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while(g[answer][c] == -1)
        answer = f[answer];
    return g[answer][c];
}

int cnt[TOTKEY];
void Matcher(vector<string> &keywords, string &text) {
    int currentState = 0;
    memset(cnt, 0, sizeof cnt);

    for(int i = 0; i < (int)text.size(); ++i) {
        currentState = findNextState(currentState, text[i]);
        if(out[currentState] == 0)                              // Nothing new, let's move
on to the next character.
            continue;

        for(int j = 0; j < (int)keywords.size(); ++j)
            if(out[currentState][j])                            // Matched keywords[j]
                ++cnt[j];
    }
}

string text, str;
vector<string>keywords;
// RETURN NUMBER OF MATHCES FOR EACH WORD APPEARING IN "KEYWORD" VECTOR
// INPUT STRING IS "TEXT"
int main() {
    int t, n;
    cin >> t;
    for(int Case = 1; Case <= t; ++Case) {
        cin >> n >> text;
        while(n--) {
            cin >> str;
            keywords.push_back(str);
        }
        buildMatchingMachine(keywords);
        cout << "Case " << Case << ":\n";
        Matcher(keywords, text);
        for(int i = 0; i < (int)keywords.size(); ++i)
            cout << cnt[i] << "\n";
        keywords.clear();
    }
    return 0;
}

// All Pair Shortest Path
// Floyd Warshal
// Complexity : O(V^3)

int G[MAX][MAX], parent[MAX][MAX];
void graphINIT() {
    for(int i = 0; i < MAX; i++)
        for(int j = 0; j < MAX; j++)
            G[i][j] = INF;
    for(int i = 0; i < MAX; i++)
        G[i][i] = 0;
}
void floydWarshall(int V) {
    for(int i = 0; i < V; i++)      // path printing matrix initialization
        for(int j = 0; j < V; j++)
            parent[i][j] = i;       // we can go to j from i by only obtaining i (by default)
    for(int k = 0; k < V; k++)      // Selecting a middle point as k
        for(int i = 0; i < V; i++)  // Selecting all combination of source (i) and destination (j)
            for(int j = 0; j < V; j++)
                if(G[i][k] != INF && G[k][j] != INF) {          // if the graph contains negative edges, then min(INF, INF+ negative edge) = +-INF!
                    G[i][j] = min(G[i][j], G[i][k]+G[k][j]);    // if G[i][i] = negative, then node i is in negative circle
                    parent[i][j] = parent[k][j];                // if path printing needed
}}
void printPath(int i, int j) {
    if(i != j) printPath(i, parent[i][j]);
    printf(" %d", j);
}
void minMax(int V) {
    for(int k = 0; k < V; k++)
        for(int i = 0; i < V; i++)
            for(int j = 0; j < V; j++)
                G[i][j] = min(G[i][j], max(G[i][k], G[k][j]));
}
void transitiveClosure(int V) {
    for(int k = 0; k < V; k++)
        for(int i = 0; i < V; i++)
            for(int j = 0; j < V; j++)
                G[i][j] |= (G[i][k] & G[k][j]);
}

// Articulation Point
// Complexity O(V+E)
// Tarjan, DFS

vector<int>G[101];
int dfs_num[101], dfs_low[101], parent[101], isAtriculationPoint[101], dfsCounter, rootChildren, dfsRoot;
void articulationPoint(int u) {
    dfs_low[u] = dfs_num[u] = ++dfsCounter;
    for(int i = 0; i < G[u].size(); i++) {
        int v = G[u][i];
        if(dfs_num[v] == 0) {
            parent[v] = u;
            if(u == dfsRoot)            // Special case for root node
                rootChildren++;         // if root node has child, increment counter
            articulationPoint(v);
            // 1 : if dfs_num[u] == dfs_low[v], then it is a back edge
            // 2 : if dfs_num[u] < dfs_low[v], then u is ancestor of v and there is no back
edge
            // so, if u is not root node, then we can chose u for Articulation Point
            if(dfs_num[u] <= dfs_low[v] && u != dfsRoot)    //Avoiding root node
                isArticulationPoint[u]++;
            // if there is any child node of u that is a back edge of a previous node
            // then the value of dfs_low[v] might be less than the present dfs_low[u]
            // we try to save the lowest value possible
            dfs_low[u] = min(dfs_low[v], dfs_low[u]);
        }
        // As nodes are bi-directional, avoiding direct child node
        // if it is not direct child node, and visited, then there is a back edge
        // so we try to decrease the value of dfs_low[u] with the dfs_num[v]
        // the dfs_num[v] is less than dfs_num[u] (as it it a back edge)
        else if(parent[u] != v)
            dfs_low[u] = min(dfs_low[u], dfs_num[v]);
}}

int main() {
    // Actual code of Articulation Point starts here
    dfsCounter = 0;
    memset(dfs_num, 0, sizeof(dfs_num));
    isArticulationPoint.reset();
    for(int i = 1; i <= n; i++) {
        if(dfs_num[i] == 0) {
            dfsCounter = rootChildren = 0;
            dfsRoot = i;
            articulationPoint(i);
            isArticulationPoint[i] = (rootChildren > 1);
        }
        // Important
        isAtriculationPoint + 1 = number of nodes that is disconnected
    }
    // Printing Articulation Points
    /*for(int i = 0; i < 101; i++)
        if(isArticulationPoint[i])
            printf("%d ", i);
    printf("\n");*/
    printf("%d\n", (int)isArticulationPoint.count());
}

// Basic BFS with path printing
// Complexity : O(V+E)

vector<int>parent, G[MAX];
void printPath(int u, int source_node) {            // destination, source
    if(u == source_node) {
        printf("%d", u);
        return;
    }
    printPath(parent[u], source_node);
    printf(" %d", u);
```

```
}

int BFS(int source_node, int finish_node, int vertices) {
    vector<int>dist(vertices+5, INF);               //contains the distance from source to end point
    queue<int>Q;
    Q.push(source_node);
    parent.resize(vertices+5, -1);                  //for path printing
    dist[source_node] = 0;

    while(!Q.empty()) {
        int u = Q.front();
        Q.pop();
        if(u == finish_node)                        //remove this line if shortest path to all nodes are needed
            return dist[u];
        for(int i = 0; i < G[u].size(); i++) {
            int v = G[u][i];
            if(dist[v] == INF) {
                dist[v] = dist[u] + 1;
                parent[v] = u;
                Q.push(v);
    }}}
    return -1;
}

int color[100];             // Contains Color (1, 2)
void Bicolor(int u) {       // Bicolor Check
    queue<int>q;
    q.push(u);
    color[u] = 1;           // Color is -1 initialized
    while(!q.empty()) {
        u = q.front();
        q.pop();
        for(int i = 0; i < (int)G[u].size(); ++i) {
            int v = G[u][i];
            if(color[v] == -1) {
                if(color[u] == 1)   color[v] = 2;
                else                color[v] = 1;
                q.push(v);
}}}}

// BigInteger By Jane Alam Jan

struct Bigint {
    string a;                   // to store the digits (in reverse order)
    int sign;                   // sign = -1 for negative numbers, sign = 1 otherwise
    Bigint() {}                 // default constructor
    Bigint(string b) {(*this) = b;}            // constructor for string
    Bigint(long long n) {
        sign = n >= 0 ? 1:-1;
        if(n == 0) {
            a.push_back('0');
            return;
        }
        while(n) {
            a.push_back(n%10 + '0');
            n /= 10;
        }
        //reverse(a.begin(), a.end());
    }
    int size() {                // returns number of digits
        return a.size();
    }
    Bigint inverseSign() {      // changes the sign
        sign *= -1;
        return (*this);
    }
    Bigint normalize(int newSign) {               // removes leading 0, fixes sign
        for( int i = a.size() - 1; i > 0 && a[i] == '0'; i-- )
            a.erase(a.begin() + i);
        sign = ( a.size() == 1 && a[0] == '0' ) ? 1 : newSign;
        return (*this);
    }
    //--------- assignment operator
    void operator = (string b) {                  // assigns a string to Bigint
        a = b[0] == '-' ? b.substr(1) : b;
        reverse(a.begin(), a.end());
        this->normalize( b[0] == '-' ? -1 : 1 );
    }
    //---------- conditional operators
    bool operator < ( const Bigint &b ) const {     // less than operator
        if( sign != b.sign ) return sign < b.sign;
        if( a.size() != b.a.size() )
            return sign == 1 ? a.size() < b.a.size() : a.size() > b.a.size();
        for( int i = a.size() - 1; i >= 0; i-- ) if( a[i] != b.a[i] )
            return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
        return false;

    }
    bool operator == ( const Bigint &b ) const {          // operator for equality
        return a == b.a && sign == b.sign;
    }
    // mathematical operators
    void Pow(int p) {                                     // Raises a Bigint to power of p
        Bigint res("1");
        while(p > 0) {
            if(p&1) res = res * (*this);
            p = p >> 1;
            (*this) = (*this) * (*this);
        }
        (*this) = res;
    }
    Bigint operator + ( Bigint b ) {                      // addition operator overloading
        if( sign != b.sign ) return (*this) - b.inverseSign();
        Bigint c;
        for(int i = 0, carry = 0; i < a.size() || i < b.size() || carry; i++) {
            carry+=(i<a.size() ? a[i]-48 : 0)+(i<b.a.size() ? b.a[i]-48 : 0);
            c.a += (carry % 10 + 48);
            carry /= 10;
        }
        return c.normalize(sign);
    }
    Bigint operator - ( Bigint b ) {                    // subtraction operator overloading
        if( sign != b.sign ) return (*this) + b.inverseSign();
        int s = sign; sign = b.sign = 1;
        if( (*this) < b ) return ((b - (*this)).inverseSign()).normalize(-s);
        Bigint c;
        for(int i = 0, borrow = 0; i < a.size(); i++ ) {
            borrow = a[i] - borrow - (i < b.size() ? b.a[i] : 48);
            c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
            borrow = borrow >= 0 ? 0 : 1;
        }
        return c.normalize(s);
    }
    Bigint operator * (Bigint b) {                          // multiplication operator
overloading
        Bigint c("0");
        for(int i = 0, k = a[i] - 48; i < a.size(); i++, k = a[i] - 48) {
            while(k--) c = c + b;                           // ith digit is k, so, we add k
times
            b.a.insert(b.a.begin(), '0');                   // multiplied by 10
        }
        return c.normalize(sign * b.sign);
    }
    Bigint operator / ( Bigint b ) {                        // division operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0"), d;
        for( int j = 0; j < a.size(); j++ ) d.a += "0";
        int dSign = sign * b.sign; b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b, d.a[i]++;
        }
        return d.normalize(dSign);
    }
    Bigint operator % ( Bigint b ) {                        // modulo operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0");
        b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b;
        }
        return c.normalize(sign);
    }
    //---------------output method
    void print() {
        if( sign == -1 ) putchar('-');
        for( int i = a.size() - 1; i >= 0; i-- ) putchar(a[i]);
}};

int main() {
    Bigint a, b, c;          // declared some Bigint variables
    string input;            // string to take input
    cin >> input;            // take the Big integer as string
    a = input;               // assign the string to Bigint a
    cin >> input;            // take the Big integer as string
    b = input;               // assign the string to Bigint
    c = a + b;             // adding a and b
    c.print();             // printing the Bigint
    puts("");              // newline
    return 0;
}

// Vertex Cover
// Wiki: Vertex Cover:
// In the mathematical discipline of graph theory, a vertex cover (sometimes node cover)
// of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set
// Wiki: Edge Cover:
// In graph theory, an edge cover of a graph is a set of edges such that every vertex of the graph
// is incident to at least one edge of the set

// Min Edge Cover = TotalNodes - MinVertexCover

bitset<MAX>vis;
int lft[MAX], rht[MAX];
vector<int>G[MAX];

int VertexCover(int u) {                    // Min Vertex Cover
    vis[u] = 1;
    for(int i = 0; i < (int)G[u].size(); ++i) {
        int v = G[u][i];
        if(vis[v]) continue;                // If v is used earlier, skip
        vis[v] = 1;
        if(lft[v] == -1) {                  // If there is no node present on left of v
            lft[v] = u, rht[u] = v;
            return 1;
        }
        else if(VertexCover(lft[v])) {      // If there is one node present on the left side of v (Let it be u')
            lft[v] = u, rht[u] = v;         // and if it is possible to match u' with another node (not v ofcourse!)
            return 1;                       // then we can match this u with v, and u' is matched with another node as well
    }}
    return 0;
}
int BPM(int n) {                            // Bipartite Matching
    int cnt = 0;
    memset(lft, -1, sizeof lft);
    memset(rht, -1, sizeof rht);
    for(int i = 1; i <= n; ++i) {           // Nodes are numbered from 1 to n
        vis.reset();
        cnt += VertexCover(i);              // Check if there exists a match for node i
    }
    return cnt;
}

//Complexity : O(V+E)
//Finding Bridges (Graph)

vector<int> G[MAX];
vector<pair<int, int> >ans;
int dfs_num[MAX], dfs_low[MAX], parent[MAX], dfsCounter;

void bridge(int u) {
    // dfs_num[u] is the dfs counter of u node
    // dfs_low[u] is the minimum dfs counter of u node (it is minimum if a backedge exists)
    dfs_num[u] = dfs_low[u] = ++dfsCounter;
    for(int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        if(dfs_num[v] == 0) {
            parent[v] = u;
            bridge(v);
            // if dfs_num[u] is lower than dfs_low[v], then there is no back edge on u node
            // so u - v can be a bridge
            if(dfs_num[u] < dfs_low[v])
                ans.push_back(make_pair(min(u, v), max(u, v)));
            // obtainig lower dfs counter (if found) from child nodes
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
        }
        // if v is not parent of u then it is a back edge
        // also dfs_num[v] must be less than dfs_low[u]
        // so we update it
        else if(parent[u] != v)
            dfs_low[u] = min(dfs_low[u], dfs_num[v]);
}}
void FindBridge(int V){                             //Bridge finding code
    memset(dfs_num, 0, sizeof(dfs_num));
    dfsCounter = 0;
    for(int i = 0; i < V; i++)
        if(dfs_num[i] == 0)
            bridge(i);
}
int main() {
    FindBridge(100);
    // Output
    sort(ans.begin(), ans.end());
    for(int i = 0; i < ans.size(); i++)
        printf("%d - %d\n", ans[i].first, ans[i].second);
    printf("\n");
    return 0;
}

vi DecimalVal(int a, int b) {           // Calculate Decimal values (after .) of a/b
    vi v;
    a %= b;
    if(a == 0) {
        v.pb(0);
        return v;
    }
    bool first = 1;
    while(SIZE(v) <= 200) {             // Define the Maximum Length of decimal values
        if(a == 0)
            return v;                   // If any Zero divisor is found (then, rest all will be Zero) return values
        else if(a < b && !first) {      // If we need to add another zero (add zero after first time)
            a*=10;
            v.pb(0);
        }
        else if(a < b && first) {       // If we need to add a extra zero (adding zero first time)
            first = 0;
            a *= 10;
            continue;
        }
        else {
            v.pb(a/b);
            a%=b;
            first = 1;
        }
    }
    return v;
}

// Repetation (PunoPonik) is also calculated
vi dec1, dec2;                              // Before . (decimal), after . (decimal)
int DecimalRepeated(int a, int b) {         // Calculate Decimal values (after .) of a/b
    unordered_map<int, int>mp;
    int k = 0, point = -1;
    bool divisable = 0;
    if(a >= b) {                            // Before Decimal Calculation
        dec1.push_back(a/b);
        a %= b;
    }
    if(dec1.size() == 0)
        dec1.push_back(0);
    while(a != 0) {
        if(mp.find(a) != mp.end()) {        // if the remainder is found again, there exists a loop
            point = mp[a];
            break;
        }
        if(a%b == 0) {
            dec2.push_back(a/b);
            break;
        }
        mp[a] = k++;
        int cnt = 0;
        while(a < b) {
            a *= 10;
            if(cnt != 0) {
                dec2.push_back(0);
                k++;
            }
            ++cnt;
        }
        if(cnt != 0 && mp.find(a) != mp.end()) {
            point = mp[a];
            break;
        }
        if(cnt == 1)
            mp[a] = (k-1);
        dec2.push_back(a/b);
        a %= b;
        if(a == 0) {
            divisable = 1;
            break;
        }}
    return divisable == 1 ? 1:((int)dec2.size()-point);
}

```

```
int main() {
    int a, b;
    cin >> a >> b;
    vi v = DecimalVal(a, b);
    for(auto it : v)
        cout << it;
    cout << endl;
    int Cycle = DecimalRepeated(a, b);
    for(auto it : dec1)
        cout << it;
    cout << ".";
    for(auto it : dec2)
        cout << it;
    cout << "\n\n";
    cout << "Last Repeating Cycle " << Cycle << endl;
    return 0;
}

// Cycle in Directed graph
// http://codeforces.com/contest/915/problem/D

vi G[550];
int color[550], Cycle = 0;      // Cycle will contain the number of cycles found in graph
void dfs(int u) {
    color[u] = 2;               // Mark as parent
    for(auto v : G[u]) {
        if(color[v] == 2)       // If any Parent found (BackEdge)
            Cycle++;
        else if(!color[v])
            dfs(v);
    }
    color[u] = 1;               // Visited
}

// Shortest Path (Dikjstra)
// Complexity : (V*logV + E)

vector<int>dist, G[MAX], W[MAX];
void printPath(int u) {     // call with ending node
    if (u == s) {           // s is the starting node
        printf("%d", s);    // base case, at the source s
        return;
    }
    printPath(p[u]);        // recursive: to make the output format: s -> ... -> t
    printf(" %d", u);
}

void dikjstra(int u, int destination, int nodes) {
    dist.resize(nodes+1, INF);                      // dist[v] contains the distance from u
to v
    dist[u] = 0;
    priority_queue<pair<int, int> > pq;             // pq is sorted in ascending order
according to weight and edge
    pq.push({0, -u});

    while(!pq.empty()) {
        int u = -pq.top().second;
        int wu = -pq.top().first;
        pq.pop();
        if(u == destination) return;                // if we only need distance of destination, then we may return
        if(wu > dist[u]) continue;                  // skipping the longer edges, if we have found shorter edge earlier

        for(int i = 0; i < G[u].size(); i++) {
            int v = G[u][i];
            int wv = W[u][i];
            if(wu + wv < dist[v]) {                 // path relax
                dist[v] = wu + wv;
                p[v] = u;                           // path printing
                pq.push({-dist[v], -v});
}}}}

// Kth Path Using Modified Dikjstra
// Complexity : O(K*(V*logV + E))
// http://codeforces.com/blog/entry/16821

vector<int>G[MAX], W[MAX], dist[MAX];
int KthDikjstra(int Start, int End, int Kth) {      // Kth Shortest Path (Visits Edge Only Once)
    for(int i = 0; i < MAX; ++i)
        dist[i].clear();
    priority_queue<pii>pq;                  // Weight, Node
    pq.push(make_pair(0, Start));

    while(!pq.empty()) {
        int u = pq.top().second;
        int w = -pq.top().first;
        pq.pop();

        if((int)dist[End].size() == Kth)    // We can also break if the Kth path is found
            return dist[End].back();
        if(dist[u].empty())
            dist[u].push_back(w);
        else if(dist[u].back() != w)        // Not taking same cost paths
            dist[u].push_back(w);           // As priority queue greedily chooses edge, it's guranteed that this edge is bigger than previous
        if((int)dist[u].size() > Kth)       // Like basic dikjstra, we'll not take the Kth+ edges
            continue;
        for(int i = 0; i < (int)G[u].size(); ++i) {
            int v = G[u][i];
            int _w = w + W[u][i];
            if((int)dist[v].size() == Kth)
                continue;
            pq.push(make_pair(-_w, v));
    }}
    return -1;
}

int KthDikjstra(int Start, int End, int Kth) {      // Kth Shortest Path (Visits Same Edge More Than Once if required)
    for(int i = 0; i < MAX; ++i)
        dist[i].clear();
    priority_queue<pii>pq;                  // Weight, Node
    pq.push(make_pair(0, Start));

    while(!pq.empty()) {
        int u = pq.top().second;
        int w = -pq.top().first;
        pq.pop();

        if(dist[u].empty())
            dist[u].push_back(w);
        else if(dist[u].back() != w) {          // if the weight is not same
            if((int)dist[u].size() < Kth)       // if we have to take more costs, take it
                dist[u].push_back(w);
            else if(dist[u].back() <= w)        // if the cost is greater than previous, then, don't go further
                continue;
            else {                              // we have to take this cost, and remove the greater one
                dist[u].push_back(w);
                sort(dist[u].begin(), dist[u].end());
                dist[u].pop_back();
        }}
        for(int i = 0; i < (int)G[u].size(); ++i) {
            int v = G[u][i];
            int _w = w + W[u][i];
            pq.push(make_pair(-_w, v));
    }}
    if((int)dist[End].size() < Kth) return -1;
    return dist[End].back();
}

// Kth Shortest Path (Every edge and shortest path of previous calculation is not used)

vector<int>G[MAX], W[MAX], S[MAX];          //edge, edge_weight, reverse_shortest_paths_graph
int dist[MAX];
bool cut_node[MAX], cut_edge[MAX][MAX];

int dikjstra(int source, int end, int nodes) {
    for(int i = 0; i < nodes; i++)          // dist[v] contains the distance from u to v
        dist[i] = INF;
    dist[source] = 0;
    priority_queue<pair<int, int> > pq;     // pq is sorted in ascending order according to weight and edge
    pq.push({0, -source});

    while(!pq.empty()) {
        int u = -pq.top().second;
        int wu = -pq.top().first;
        pq.pop();
        if(wu > dist[u]) continue;          // skipping the longer edges, if we have found shorter edge earlier

        for(int i = 0; i < (int)G[u].size(); i++) {
            int v = G[u][i];
            int wv = W[u][i];

            if(cut_node[v] || cut_edge[u][v])   // if there exists node/edge that is used in previous shortest path
                continue;
```

```cpp
            if(wu + wv < dist[v]) {           // path relax
                dist[v] = wu + wv;
                S[v].clear();                   // if this edge is smaller than other edge, then we refresh the reverse paths of this node
                S[v].push_back(u);              // then push back the node, (building a reverse graph of shortest path(s) )
                pq.push({-dist[v], -v});
            }
            else if(wu + wv == dist[v])         // if there is more than one shortest paths, then only add it in the reverse graph, nothing else
                S[v].push_back(u);
    }}
    return dist[end];
}

void cut_off(int start, int destination) {      // this function cuts off all the nodes
    if(destination == start) return;
    for(int i = 0; i < S[destination].size(); i++) {
        int v = S[destination][i];
        cut_node[v] = 1;
        cut_edge[destination][v] = cut_edge[v][destination] = 1;
        cut_off(start, v);
}}

//-----------------------String DP-----------------------
int Palindrome(int l, int r) {               // Building Palindrome in minimum move
    if(dp[l][r] != INF) return dp[l][r];
    if(l >= r)          return dp[l][r] = 0;
    if(l+1 == r)        return dp[l][r] = (s[l] != s[r]);
    if(s[l] == s[r])    return dp[l][r] = Palindrome(l+1, r-1);
    return dp[l][r] =   min(Palindrome(l+1, r), Palindrome(l, r-1))+1;      // Adding a alphabet on right, left
}

void dfs(int l, int r) {                // Palindrome printing, for above DP function
    if(l > r) return;
    if(s[l] == s[r]) {
        Palin.push_back(s[l]);
        dfs(l+1, r-1);
        if(l != r) Palin.push_back(s[l]);
        return;
    }
    int P = min(make_pair(dp[l+1][r], 1), make_pair(dp[l][r-1], 2)).second;
    if(P == 1) {
        Palin.push_back(s[l]);
        dfs(l+1, r);
        Palin.push_back(s[l]);
    }
    else {
        Palin.push_back(s[r]);
        dfs(l, r-1);
        Palin.push_back(s[r]);
}}

bool isPalindrome(int l, int r) {       // Checks if substring l-r is palindrome
    if(l == r || l > r)     return 1;
    if(dp[l][r] != -1)      return dp[l][r];
    if(s[l] == s[r])        return dp[l][r] = isPalindrome(l+1, r-1);
    return 0;
}

int recur(int p1, int p2) {         // make string s1 like s2, in minimum move
    if(dp[p1][p2] != INF)
        return dp[p1][p2];
    if(p1 == l1 or p2 == l2) {      // reached end of string s1 or s2
        if(p1 < l1) return dp[p1][p2] = recur(p1+1, p2)+1;
        if(p2 < l2) return dp[p1][p2] = recur(p1, p2+1)+1;
        return dp[p1][p2] = 0;
    }
    if(s1[p1] == s2[p2])            // match found
        return dp[p1][p2] = recur(p1+1, p2+1);
    // change at position p1, delete position p1, insert at position p1
    return dp[p1][p2] = min(recur(p1+1, p2+1), min(recur(p1+1, p2), recur(p1, p2+1)))+1;
}

void dfs(int p1, int p2) {          // printing function for above dp
    if(dp[p1][p2] == 0)             // end point (value depends on topdown/bottomup)
        return;
    if(s1[p1] == s2[p2]) {          // match found, no operation
        dfs(p1+1, p2+1);
        return;
    }
    int P = min(mp(dp[p1+1][p2], 1), min(mp(dp[p1][p2+1], 2), mp(dp[p1+1][p2+1],
3))).second;
    if(P == 1)      dfs(p1+1, p2);              // delete s1[p1] from position p2 of s1
string
    else if(P == 2) dfs(p1, p2+1);              // insert s2[p2] on position p2 of s1 string
    else            dfs(p1+1, p2+1);            // change s1[p2] to s2[p2] on position p2 of
string s1
}

int reduce(int l, int r) {              // Reduce string AXDOODOO (len : 8) to AX(DO^2)^2
(len : 4)
    if(l > r)           return INF;
    if(l == r)          return 1;
    if(dp[l][r] != -1)  return dp[l][r];
    int ret = r-l+1;
    int len = ret;
    for(int i = l; i < r; ++i)          // A B D O O D O O   remove A X substring
        ret = min(ret, reduce(l, i)+reduce(i+1, r));
    for(int d = 1; d < len; ++d) {      // D O O D O O  to check all divisable length
substring
        if(len%d != 0)  continue;
        for(int i = l+d; i <= r; i += d)
            for(int k = 0; k < d; ++k)
                if(s[l+k] != s[i+k])
                    goto pass;
        ret = min(ret, reduce(l, l+d-1));
        pass:;
    }
    return dp[l][r] = ret;
}

// Light OJ 1073 - DNA Sequence
// FIND and PRINT shortest string after merging multiple string together

```

```
int matchDP[20][20];
int TryMatch(int x, int y) {            // Finds First overlap of two string
    if(matchDP[x][y] != -1)             // ABAAB + AAB : Match at 2
        return matchDP[x][y];
    for(size_t i = 0; i < v[x].size(); ++i) {
        for(size_t j = i, k = 0; j < v[x].size() && k < v[y].size(); ++j, ++k)
            if(v[x][j] != v[y][k])
                goto pass;
        return matchDP[x][y] = i;
        pass:;
    }
    return matchDP[x][y] = v[x].size();
}

int dp[16][(1<<15)+100];
int recur(int mask, int last) {                     // DP part of LIGHT OJ
    if(dp[last][mask] != -1)                        // eleminate all substrings from n string
first in main function!
        return dp[last][mask];                      // it's not handeled here
    if(mask == (1<<n)-1)
        return dp[last][mask] = v[last].size();
    int ret = INF, cost;
    for(int i = 0; i < n; ++i) {
        if(isOn(mask, i))
            continue;
        int mPos = TryMatch(last, i);
        if(mPos < (int)v[last].size())
            cost = (int)v[last].size() - ((int)v[last].size() - mPos);
        else
            cost = v[last].size();
        ret = min(ret, recur(mask | (1 << i), i) + cost);
    }
    return dp[last][mask] = ret;
}

string ans;
void dfs(int mask, int last, string ret) {          // PRINTING part of LIGHT OJ
    if(!ret.empty() && ans < ret)
        return;
    if(mask == (1<<n)-1) {
        ret += v[last];
        if(ret < ans)
            ans = ret;
        return;
    }
    for(int i = 0; i < n; ++i) {
        if(isOn(mask, i))
            continue;
        int mPos = TryMatch(last, i);
        int cost;
        if(mPos < (int)v[last].size())
            cost = (int)v[last].size() - ((int)v[last].size() - mPos);
        else
            cost = v[last].size();
        if(dp[last][mask] - cost == dp[i][mask | (1<<i)])
            dfs(mask | (1<<i), i, ret + v[last].substr(0, cost));
}}

```

```
//-----------------------Digit DP-----------------------

// Complexity : O(10*idx*sum*tight)     : LightOJ 1068
// Tight contains if there is any restriction to number (Tight is initially 1)
// Initial Params: (MaxDigitSize-1, 0, 0, 1, modVal, allowed_digit_vector)

ll dp[15][100][100][2];
ll digitSum(int idx, int sum, ll value, bool tight, int mod, vector<int>&MaxDigit) {
    if (idx == -1)
        return ((value == 0) && (sum == 0));
    if (dp[idx][sum][value][tight] != -1)
        return dp[idx][sum][value][tight];
    ll ret = 0;
    int lim = (tight)? MaxDigit[idx] : 9;                           // Numbers are genereated in reverse order
    for (int i = 0; i <= lim; i++) {
        bool newTight = (MaxDigit[idx] == i)? tight : 0;            // caclulating newTight value for next state
        ll newValue = value ? ((value*10) % mod)+i : i;
        ret += digitSum(idx-1, (sum+i)%mod, newValue%mod, newTight, mod, MaxDigit);
    }
    return dp[idx][sum][value][tight] = ret;
}

// Bit DP (Almost same as Digit DP)    : LighOJ 1032
// Complexity O(2*pos*total_bits*tights*number_of_bits)
// Initial Params : (MostSignificantOnBitPos, N, 0, 0, 1)
// Call as : bitDP(SigOnBitPos, N, 0, 0, 1)   N is the Max Value, calculating [0 - N]
// Tight is initially on
// pairs are number of paired bits, prevOn shows if previous bit was on (it is for this problem)

#define isOn(x, i) (x & (1LL<<i))
#define On(x, i) (x | (1LL<<i))
#define Off(x, i) (x & ~(1LL<<i))
int N, lastBit;
long long dp[33][33][2][2];
ll bitDP(int pos, int mask, int pairs, bool prevOn, bool tight) {
    if(pos < 0)
        return pairs;
    if(dp[pos][pairs][prevOn][tight] != -1)
        return dp[pos][pairs][prevOn][tight];
    bool newTight = tight & !isOn(mask, pos);    // Turn off tight when we are turning off a bit which was initially on
    ll ans = bitDP(pos-1, Off(mask, pos), pairs, 0, newTight);
    if(On(mask, pos) <= N)
        ans += bitDP(pos-1, On(mask, pos), pairs + prevOn, 1, tight && isOn(mask, pos));
    return dp[pos][pairs][prevOn][tight] = ans;
}

// Memory Optimized DP + Bottom Up solution (LOJ : 1126 - Building Twin Towers)
// given array v of n elements, make two value x1 and x2 where x1 == x2, output maximum of it

int dp[2][500010], n;                                   // present dp table and past dp table
int BottomUp(int TOT) {                                 // TOT = (Cumulative Sum of v)/2
    memset(dp, -1, sizeof dp);
    dp[0][0] = 0;
    bool present = 0, past = 1;
    for(int i = 0; i < n; ++i) {
        present ^= 1, past ^= 1;                        // Swapping present and past dp table
        for(int diff = 0; diff <= TOT; ++diff)
            if(dp[past][diff] != -1) {
                int moreDiff = diff + v[i], lessDiff = abs(diff - v[i]);
                dp[present][diff] = max(dp[present][diff], dp[past][diff]);
                dp[present][lessDiff] = max(dp[present][lessDiff], max(dp[past][lessDiff], dp[past][diff] + v[i]));
                dp[present][moreDiff] = max(dp[present][moreDiff], max(dp[past][moreDiff], dp[past][diff] + v[i]));
    }}
    return (max(dp[0][0], dp[1][0]))/2;                 // Returns the maximum possible answer
}

// Count Number of ways to go from (1, 1) to (r, c) if there exists n unassassable points (only eight and down is valid move)
ll CountNumberofWays(int r, int c, int n) {
    v[n] = {r, c};                                  // also add the last point as unaccessable point, to find how many
    sort(v.begin(), v.end());                       // ways we can come to this point, which is the answer
    for(int i = 0; i <= n; ++i) {
        dp[i] = CountWay(1, 1, v[i].first, v[i].second);               // Number of ways we can come from starting square
        for(int j = 0; j < i; ++j)
            if(v[j].first <= v[i].first and v[j].second <= v[i].second)
                dp[i] = (dp[i] - (dp[j] * CountWay(v[j].first, v[j].second, v[i].first, v[i].second))%MOD + MOD)%MOD;
    }                                               // Number of ways we can reach from (1, 1) to (r, c)
    return dp[n];                                   // The last state is always (r, c), which is the answer
}

// Travelling Salesman
// dist[u][v] = distance from u to v
// dp[u][bitmask] = dp[node][set_of_taken_nodes]  (saves the best(min/max) path)
// call with tsp(starting node, 1)

int n, x[11], y[11], dist[11][11], memo[11][1 << 11], dp[11][1 << 11];
int TSP(int u, int bitmask) {                   // startin node and bitmask of taken nodes
    if(bitmask == ((1 << (n)) - 1))             // when it steps in this node, if all nodes are visited
        return dist[u][0];                      // then return to 0'th (starting) node [as the path is hamiltonian]
    // or use return dist[u][start] if starting node is not 0
    if(dp[u][bitmask] != -1)                    // if we have previous value set up
        return dp[u][bitmask];                  // use that previous value
    int ans = 1e9;                              // set an infinite value
    for(int v = 0; v <= n; v++)                     // for all the nodes
        if(u != v && !(bitmask & (1 << v)))         // if this node is not the same node, and if this node is not used yet(in bitmask)
            ans = min(ans, dist[u][v] + tsp(v, bitmask | (1 << v)));    // min(past_val, dist u->v + dist(v->all other untaken nodes))
    return dp[u][bitmask] = ans;                // save in dp and return
}

// Basic DSU with compression

struct DSU {
    vector<int>u_list, u_set;         // u_list[x] : the size of a set x,  u_set[x] : the root of x
    DSU() {}
    DSU(int SZ) { init(SZ); }
    int unionRoot(int n) {                          // Union making with dynamic compression
        if(u_set[n] == n) return n;
        return u_set[n] = unionRoot(u_set[n]);      // Directly set the actual root of this set as root (Compress)
    }
    int makeUnion(int a, int b) {                   // Union making with compression
        int x = unionRoot(a), y = unionRoot(b);
        if(x == y) return x;                        // If both are in same set
        else if(u_list[x] > u_list[y]) {            // Makes x root (y -> x)
            u_set[y] = x;
            u_list[x] += u_list[y];                 // Root's size is increased
            return x;
        }
        else {                                      // Makes y root (x -> y)
            u_set[x] = y;
            u_list[y] += u_list[x];                 // Root's size is increased
            return y;
    }}
    void init(int len) {                            // Initializer
        u_list.resize(len+5);
        u_set.resize(len+5);
        for(int i = 0; i <= len+3; i++)
            u_set[i] = i, u_list[i] = 1;            // Each node contains itself, so size of each node set to 1
    }
    bool isRoot(int x) {                            // Returns true if this is a root (May contain one or many nodes)
        return u_set[x] == x;
    }
    bool isRootContainsMany(int x) {                // If the root contains more than one value (Actual Root)
        return (isRoot(x) && (u_list[x] > 1));
    }
    bool isSameSet(int a, int b) {                  // If a and b is in same set/component
        return (unionRoot(a) == unionRoot(b));
}};

// Bipartite DSU (Tested)

struct BipartiteDSU {
    vector<int>u_list, u_set, u_color;
    vector<bool>missmatch;                                  // Bicolor missmatch

    BipartiteDSU() {}
    BipartiteDSU(int SZ) { init(SZ); }

    pll unionRoot(int n) {                                  // Union making with dynamic compression
```

```
        if(u_set[n] == n) return {n, u_color[n]};
        pll root = unionRoot(u_set[n]);
        if(missmatch[u_set[n]] or missmatch[n])
            missmatch[n] = missmatch[u_set[n]] = 1;
        u_color[n] = (u_color[n] + root.second)&1;
        u_set[n] = root.first;                              // Directly set the actual root of this set as root (Compress)
        return {u_set[n], u_color[n]};
    }
    int makeUnion(int a, int b) {                                   // Union making with compression
        int x = unionRoot(a).first, y = unionRoot(b).first;
        if(x == y) {                                                // If both are in same set and bipartite missmatch exists
            if(u_color[a] == u_color[b]) missmatch[x] = 1;
            return x;
        }
        if(missmatch[x] or missmatch[y])                            // Checks if Bipartite missmatch exists
            missmatch[x] = missmatch[y] = 1;
        if(u_list[x] < u_list[y]) {                                 // Makes x root (y -> x)
            u_set[x] = y;
            u_list[x] += u_list[y];                                 // Root's size is increased
            u_color[x] = (u_color[a]+u_color[b]+1)&1;               // Setting color of component y according to the color of a & b
            return y;
        }
        else {                                          // Makes y root (x -> y)
            u_set[y] = x;
            u_list[y] += u_list[x];                     // Root's size is increased
            u_color[y] = (u_color[a]+u_color[b]+1)&1;   // Setting color of component y according to the color of a & b
            return x;
    }}
    void init(int len) {                // Initializer
        u_list.resize(len+5);
        u_set.resize(len+5);
        u_color.resize(len+5);
        missmatch.resize(len+5);
        for(int i = 0; i <= len+3; i++)
            u_set[i] = i, u_list[i] = 1, u_color[i] = 0, missmatch[i] = 0;
    }
    bool isRoot(int x) {                // Returns true if this is a root (May contain one or many nodes)
        return u_set[x] == x;
    }
    bool isRootContainsMany(int x) {    // If the root contains more than one value (Actual Root)
        return (isRoot(x) && (u_list[x] > 1));
    }
    bool isSameSet(ll a, ll b) {        // If a and b is in same set/component
        return (unionRoot(a).first == unionRoot(b).first);
    }
    int getColor(ll u) {                // Color of node u (DONT get the color of root)
        return u_color[u];
    }
    bool hasMissmatch(int x) {          // If there is bipartite missmatch in this set/component
        return missmatch[x];
}};

// Dynamic Weighted DSU (Checked, Not Tested)

struct WeightedDSU {
    vector<int>u_list, u_set, u_weight, weight;
    WeightedDSU() {}
    WeightedDSU(int SZ) { init(SZ); }
    int unionRoot(int n) {                                  // Union making with compression
        if(u_set[n] == n) return n;
        return u_set[n] = unionRoot(u_set[n]);              // Directly set the actual root of this set as root (Compress)
    }
    void changeWeight(int u, int w, bool first = 1) {       // Change any component's weight (Dynamic)
        if(first) w = w - weight[u];
        u_weight[u] += w;
        if(u_set[u] != u)
            changeWeight(u_set[u], w, 0);
    }
    int makeUnion(int a, int b) {                           // Union making with compression
        int x = unionRoot(a), y = unionRoot(b);
        if(x == y) return x;
        if(u_list[x] > u_list[y]) {                         // Makes x root (y -> x)
            u_set[y] = x;
            u_list[x] += u_list[y];                         // Root's size is increased
            u_weight[x] += u_weight[y];                     // Root's weight is increased
            return x;
        }
        else {                                              // Makes y root (x -> y)
            u_set[x] = y;
            u_list[y] += u_list[x];                         // Root's size is increased
            u_weight[y] += u_weight[x];                     // Root's weight is increased
            return y;
    }}
    void init(int len) {                                    // Initializer
        u_list.resize(len+5);
        u_set.resize(len+5);
        u_weight.resize(len+5);
        weight.resize(len+5);
        for(int i = 0; i <= len+3; i++)
            u_set[i] = i, u_list[i] = 1, u_weight[i] = weight[i] = 0;
    }
    bool isRoot(int x) {                // Returns true if this is a root (May contain one or many nodes)
        return u_set[x] == x;
    }
    bool isRootContainsMany(int x) {            // If the root contains more than one value (Actual Root)
        return (isRoot(x) && (u_list[x] > 1));
    }
    bool isSameSet(int a, int b) {              // If a and b is in same set/component
        return (unionRoot(a) == unionRoot(b));
    }
    void setWeight(int u, int w) {              // Set weight of node u to w, run before union
        u_weight[u] = w;
        weight[u] = w;
    }
    int getComponentWeight(int u) {             // Get weight sum of the set/comopnent
        return u_weight[unionRoot(u)];
}};

// 1D Fenwick Tree

struct BIT {
    ll tree[MAX];
    int MaxVal;
    void init(int sz=1e7) {
        memset(tree, 0, sizeof tree);
        MaxVal = sz+1;
    }
    void update(int idx, ll val) {
        for( ;idx <= MaxVal; idx += (idx & -idx))
            tree[idx] += val;
    }
    void update(int l, int r, ll val) {
        if(l > r) swap(l, r);
        update(l, val);
        update(r+1, -val);
    }
    ll read(int idx) {
        ll sum = 0;
        for( ;idx > 0; idx -= (idx & -idx))
            sum += tree[idx];
        return sum;
    }
    ll read(int l, int r) {
        ll ret = read(r) - read(l-1);
        return ret;
    }
    ll readSingle(int idx) {             // Point read in log(n)
        ll sum = tree[idx];
        if(idx > 0) {
            int z = idx - (idx & -idx);
            --idx;
            while(idx != z) {
                sum -= tree[idx];
                idx -= (idx & -idx);
        }}
        return sum;
    }
    int search(int cSum) {
        int pos = -1, lo = 1, hi = MaxVal, mid;
        while(lo <= hi) {
            mid = (lo+hi)/2;
            if(read(mid) >= cSum) {     // read(mid) >= cSum : to find the lowest index of cSum value
                pos = mid;              // read(mid) == cSum : to find the greatest index of cSum value
                hi = mid-1;
            }
            else
                lo = mid+1;
        }
        return pos;
    }
    ll size() {
        return read(MaxVal);
    }
    // Modified BIT, this section can be used to add/remove/read 1 to all elements from 1 to pos
    // all of the inverse functions must be used, for any manipulation
    ll invRead(int idx) {           // gives summation from 1 to idx
        return read(MaxVal-idx);
    }
    void invInsert(int idx) {       // adds 1 to all index less than idx
        update(MaxVal-idx, 1);
    }
    void invRemove(int idx) {       // removes 1 from idx
        update(MaxVal-idx, -1);
    }
    void invUpdate(int idx, ll val) {
        update(MaxVal-idx, val);
    }
}};
// --------------------------- 2D Fenwick Tree -------------------------
/* /\
 y  |
    |   (x1,y2) -------- (x2,y2)
    |          |       |
    |          |       |
    |          |       |
    |          ---------
    |   (x1,y1)         (x2, y1)
    |
    |___________________________
   (0, 0)                   x-->        */

ull tree[2510][2510];
int xMax = 2505, yMax = 2505;
// Updates from min point to MAX LIMIT
void update(int x, int y, ll val) {
    int y1;
    while(x <= xMax) {
        y1 =  y;
        while(y1 <= yMax) {
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
}}
ll read(int x, int y) {         // Reads from (0, 0) to (x, y)
    ll sum = 0;
    int y1;
    while(x > 0) {
        y1 = y;
        while(y1 > 0) {
            sum += tree[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return sum;
}
ll readSingle(int x, int y) {
    return read(x, y) + read(x-1, y-1) - read(x-1, y) - read(x, y-1);
}
void updateSquare(pii p1, pii p2, ll val) {     // p1 : lower left point, p2 : upper right
point
    update(p1.first, p1.second, val);
    update(p1.first, p2.second+1, -val);
    update(p2.first+1, p1.second, -val);
    update(p2.first+1, p2.second+1, val);
}
ll readSquare(pii p1, pii p2) {                // p1 : lower left point, p2 : upper right
point
    ll ans = read(p2.first, p2.second);
    ans -= read(p1.first-1, p2.second);
    ans -= read(p2.first, p1.second-1);
    ans += read(p1.first-1, p1.second-1);
    return ans;
}

// // --------------------------- 3D Fenwick Tree -------------------------

ll tree[105][105][105];
ll xMax = 100, yMax = 100, zMax = 100;
void update(int x, int y, int z, ll val) {
    int y1, z1;
    while(x <= xMax) {
        y1 = y;
        while(y1 <= yMax) {
            z1 = z;
            while(z1 <= zMax) {
                tree[x][y1][z1] += val;
                z1 += (z1 & -z1);
            }
            y1 += (y1 & -y1);
        }
        x += (x & -x);
}}
ll read(int x, int y, int z) {
    ll sum = 0;
    int y1, z1;
    while(x > 0) {
        y1 = y;
        while(y1 > 0) {
            z1 = z;
            while(z1 > 0) {
                sum += tree[x][y1][z1];
                z1 -= (z1 & -z1);
            }
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return sum;
}
```

```
ll readRange(ll x1, ll y1, ll z1, ll x2, ll y2, ll z2) {
    --x1, --y1, --z1;
    return  read(x2, y2, z2)
        - read(x1, y2, z2)
        - read(x2, y1, z2)
        - read(x2, y2, z1)
        + read(x1, y1, z2)
        + read(x1, y2, z1)
        + read(x2, y1, z1)
        - read(x1, y1, z1);
}
void updateRange(int x1, int y1, int z1, int x2, int y2, int z2) {      // Not tested yet!!
    update(x1, y1, z1, val);
    update(x2+1, y1, z1, -val);
    update(x1, y2+1, z1, -val);
    update(x1, y1, z2+1, -val);
    update(x2+1, y2+1, z1, val);
    update(x1, y2+1, z2+1, val);
    update(x2+1, y1, z2+1, val);
    update(x2+1, y2+1, z2+1, -val);
}
// Pattens to built BIT update read:
// always starts with first(starting point), add val
// take (1 to n) elements from ending point with all combination add it to staring point,
add (-1)^n * val

struct fraction {                    // Fraction Calculation Template
    int a, b;
    fraction() {
        a = 1;
        b = 1;
    }
    fraction(int x, int y) : a(x), b(y) {}
    flip() {swap(a, b);}
    fraction operator + (fraction other) {
        fraction temp;
        temp.b = ((b)*(other.b))/(__gcd((b), other.b));
        temp.a = (temp.b/b)*a + (temp.b/other.b)*other.a;
        int x = __gcd(temp.a, temp.b);
        if(x != 1) {temp.a/=x; temp.b/=x;}
        return temp;
    }
    fraction operator - (fraction other) {
        fraction temp;
        temp.b = (b*other.b)/__gcd(b, other.b);
        temp.a = (temp.b/b)*a - (temp.b/other.b)*other.a;
        int x = __gcd(temp.a, temp.b);
        if(x != 1) {temp.a/=x; temp.b/=x;}
        return temp;
    }
    fraction operator / (fraction other) {
        fraction temp;
        temp.a = a*other.b;
        temp.b = b*other.a;
        int x = __gcd(temp.a, temp.b);
        if(x != 1) {temp.a/=x; temp.b/=x;}
        return temp;
    }
    fraction operator * (fraction other) {
        fraction temp;
        temp.a = a*other.a;
        temp.b = b*other.b;
        int x = __gcd(temp.a, temp.b);
        if(x != 1) {temp.a/=x; temp.b/=x;}
        return temp;
}};

struct BaseInt {                                        // Number Base Conversions
    string val;
    ll base;
    BaseInt() {}
    BaseInt(string _val, int _base = 10) {          // Do check if any value if val is
greater than base
        val = _val;                                 // Which is impossible
        base = _base;
    }
    char reVal(int num) {
        if(num >= 0 && num <= 9) return (char)(num + '0');
        return (char)(num - 10 + 'A');
    }
    int getVal(char c) {
        if(c <= '9' && c >= '0') return c-'0';
        return c-'A'+10;
    }
    void DecimalTo(int _base) {
        ll v = stoll(val);
        base = _base;
        val.clear();
        while(v) {
            val.push_back(reVal(v%base));
            v /= base;
        }
        reverse(val.begin(), val.end());
        if(val.empty()) val.push_back('0');
    }
    bool ToDecimal() {
        ll ret = 0;
        for(int i = 0; i < (int)val.size(); ++i) {
            int v = getVal(val[i]);
            if(v >= base)       return 0;
            if(i)               ret *= base;
            ret += v;
        }
        val = to_string(ret), base = 10;
        return 1;
    }
    void ChangeBase(int to) {
        if(base == to)  return;             // If input is "000", then output will also be
"000" (if base remains same)
        if(base != 10)  ToDecimal();        // remove the if statements to recover
        if(to != 10)    DecimalTo(to);
    }
    void Reverse() {
        reverse(val.begin(), val.end());
    }
    BaseInt operator + (BaseInt other) const {
        BaseInt a(val, base), b = other;
        a.ToDecimal(), b.ToDecimal();
        string sum = to_string(stoi(a.val, 0) + stoi(b.val, 0));
        BaseInt ret(sum);
        ret.ChangeBase(base);
        return ret;
}};

// The trig functions of C++ take radian exclusively
// 0D Objects---------------------------------------------------------------
struct point {          // In Integer
    int x, y;
    point() { x = y = 0; }
    point(int _x, int _y) : x(_x), y(_y) {}

    bool operator < (point other) const {
        if(x != other.x)
            return x < other.x;
        return y < other.y;
    }

    bool operator == (point other) const {
        return (x == other.x) && (y == other.y);
}};

struct point {          // In Double
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}

    bool operator < (point other) const {
        if (fabs(x - other.x) > EPS)
            return x < other.x;
        return y < other.y;
    }
    bool operator == (point other) const {
        return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS));
}};

bool Equal(double a, double b) {
    return (fabs(a-b) <= EPS);
}

int hypot(point p1, point p2) {
    int x = p1.x-p2.x;
    int y = p1.y-p2.y;
    return x*x + y*y;
}

double dist(point p1, point p2) {
    int x = p1.x-p2.x;
    int y = p1.y-p2.y;
    return sqrt(x*x + y*y);
}

double DEG_to_RAD(double deg) {             // Converts Degree to Radian
    return (deg*PI)/180;
}
```

```

double RAD_to_DEG(double rad) {
    return (180/PI)*rad;
}

point rotate(point p, double theta) {       // Rotates point p w.r.t. origin.  (theta is in degree)
    double rad = DEG_to_RAD(theta);
    return point(p.x * cos(rad) - p.y * sin(rad), p.x * sin(rad) + p.y * cos(rad));
}

double PointToArea(point p1, point p2, point p3) {                      // Returns Positive Area in if the points are clockwise, Negative for Anti-Clockwise
    return (p1.x*(p2.y-p3.y) + p2.x*(p3.y-p1.y) + p3.x*(p1.y-p2.y));    // Divide by 2 if Triangle area is needed
}

double whichSide(point p, point q, point r) {     // returns on which side point r is w.r.t pq line
    double slope = (p.y-q.y)*(q.x-r.x) - (q.y-r.y)*(p.x-q.x);
    return slope;                          // slope = 0 : linear, slope > 0 : right, slope < 0 : left
}

// 1D Objects-------------------------------------------------------------
struct line {
    double a, b, c;
};
void pointsToLine(point p1, point p2, line &l) {    // ax + by + c = 0  [comes from y = mx + c]
    if (fabs(p1.x - p2.x) < EPS)                    // vertical line is fine
        l.a = 1.0, l.b = 0.0, l.c = -p1.x;          // default values
    else {
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0;                                  // IMPORTANT: we fix the value of b to 1.0
        l.c = -(double)(l.a * p1.x) - p1.y;
}}
bool areParallel(line l1, line l2) {            // check coefficients a & b
    return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS);
}
bool areSame(line l1, line l2) {                // also check coefficient c
    return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS);
}
bool areIntersect(line l1, line l2, point &p) {
    if(areParallel(l1, l2)) return 0;                                   // no intersection
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);    // solve system of 2 linear algebraic equations with 2 unknowns
    if(fabs(l1.b) > EPS)    p.y = -(l1.a * p.x + l1.c);                 // special case: test for vertical line to avoid division by zero
    else                    p.y = -(l2.a * p.x + l2.c);
    return 1;
}
line perpendicularLine(line l, point p) {           // returns a perpendicular line on l which goes throuth
    line ret;                                       // point p
    ret.a = l.b, ret.b = -l.a;
    ret.c = -(ret.a*p.x + ret.b*p.y);
    if(ret.b < 0) ret.a *= -1, ret.b *= -1, ret.c *= -1;    // as line must contain b = 1.0 by default
    if(ret.b != 0) {
        ret.a /= ret.b;
        ret.c /= ret.b;
        ret.b = 1;
    }
    return ret;
}

// Vectors ------------------------------
struct vec {
    double x, y;                        // name: ‘vec’ is different from STL vector
    vec(double _x, double _y) : x(_x), y(_y) {}
};
vec toVec(point a, point b) {           // convert 2 points to vector a->b
    return vec(b.x - a.x, b.y - a.y);
}
vec scale(vec v, double s) {            // nonnegative s = [<1 .. 1 .. >1]
    return vec(v.x * s, v.y * s);       // shorter.same.longer
}
point translate(point p, vec v) {       // translate p according to v
    return point(p.x + v.x , p.y + v.y);
}
double dot(vec a, vec b) {
    return (a.x * b.x + a.y * b.y);
}
double norm_sq(vec v) {
    return v.x * v.x + v.y * v.y;
}

// Parametric Line ------------------------------
struct ParaLine {                                       // Line in Parametric Form
    point a, b;                                         // points must be in DOUBLE
    ParaLine() { a.x  = a.y = b.x = b.y = 0; }
    ParaLine(point _a, point _b) : a(_a), b(_b) {}      // {Start, Finish} or {from, to}

    point getPoint(double t) {                          // Parametric Line : a + t * (b - a) t = [-inf, +inf]
        return point(a.x + t*(b.x-a.x), a.y + t*(b.y-a.y));
}};

// Returns the distance from p to the line defined by  two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {        // formula: c = a + u * ab
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    c = translate(a, scale(ab, u));                             // translate a to c
    return dist(p, c);                                          // Euclidean distance between p and c
}

// Returns the distance from p to the line segment ab defined by two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    if(u < 0.0) {
        c = point(a.x, a.y);        // closer to a
        return dist(p, a);          // Euclidean distance between p and a
    }
    if (u > 1.0) {
        c = point(b.x, b.y);        // closer to b
        return dist(p, b);          // Euclidean distance between p and b
    }
    return distToLine(p, a, b, c);  // run distToLine as above
}

// Returns the angle aob given three points: a, o, and b, (using dot product)
double angle(point a, point o, point b) {       // returns angle aob in rad
    vec oa = toVec(o, a), ob = toVec(o, b);
    return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

double cross(vec a, vec b) {                    // Cross product of two vectors
    return a.x * b.y - a.y * b.x;               // note: to accept collinear points, we have
to change the ‘> 0’
}

bool ccw(point p, point q, point r) {           // returns true if point r is on the left
side of line pq
    return cross(toVec(p, q), toVec(p, r)) > 0;
}

bool collinear(point p, point q, point r) {     // returns true if point r is on the same
line as the line pq
    return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}

// 2D Objects ---------------------------------------------------------------
// --------- CIRCLE ---------
struct circle {
    int x, y, r;
    circle(int _x, int _y, int _r) {
        x = _x;
        y = _y;
        r = _r;
    }
    double Area() {
        return PI*r*r;
    }
};

// Reference: https://www.mathsisfun.com/geometry/circle-sector-segment.html
double CircleSegmentArea(double r, double theta) {      // Circle Radius, Center
Angle(degree)
    return r * r * 0.5 * (DEG_to_RAD(theta) - sin(DEG_to_RAD(theta)));
}
double CircleSectorArea(double r, double theta) {       // Circle Radius, Center
Angle(degree)
    return r * r * 0.5 * DEG_to_RAD(theta);
}
double CircleArcLength(double r, double theta) {        // Circle Radius, Center
Angle(degree)
```

```
    return r * DEG_to_RAD(theta);
}
bool doIntersectCircle(circle c1, circle c2) {
    int dis = dist(point(c1.x, c1.y), point(c2.x, c2.y));
    if(sqrt(dis) < c1.r+c2.r) return 1;
    return 0;
}
bool isInside(circle c1, circle c2) {                   // Returns true if any one of the circle is fully into another circle
    int dis = dist(point(c1.x, c1.y), point(c2.x, c2.y));
    return ((sqrt(dis) <= max(c1.r, c2.r)) and (sqrt(dis) + min(c1.r, c2.r) < max(c1.r, c2.r)));
}
// Returns where a point p lies according to a circle of center c and radius r
int insideCircle(point p, point c, int r) {         // all integer version
    int dx = p.x - c.x, dy = p.y - c.y;
    int Euc = dx * dx + dy * dy, rSq = r * r;     // all integer
    return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;      // inside(0)/border(1)/outside(2)
}

// Given 2 points on the circle (p1 and p2) and radius r of the corresponding circle,
// determine the location of the centers (c1 and c2) of the two possible circles
bool circle2PtsRad(point p1, point p2, double r, point &c) {
    double d2 = (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
    double det = r * r / d2 - 0.25;
    if(det < 0.0) return false;
    double h = sqrt(det);
    c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
    c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
    return true;                                        // to get the other center, reverse p1 and p2
}

// --------- Triangle ---------
double TriangleArea(double AB, double BC, double CA) {
    double s = (AB + BC + CA)/2.0;
    return sqrt(s*(s-AB)*(s-BC)*(s-CA));
}
double getAngle(double AB, double BC, double CA) {      // Returns the angle(IN RADIAN) opposite of side CA
    return acos((AB*AB + BC*BC - CA*CA)/(2*AB*BC));
}
double rInCircle(double ab, double bc, double ca) {     // Returns radius of inCircle of a triangle
    return TriangleArea(ab, bc, ca) / (0.5 * (ab + bc+ ca));
}
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
    r = rInCircle(p1, p2, p3);
    if (fabs(r) < EPS) return 0;                    // no inCircle center
    line l1, l2;
    double ratio = dist(p1, p2) / dist(p1, p3);     // compute these two angle bisectors
    point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
    pointsToLine(p1, p, l1);
    ratio = dist(p2, p1) / dist(p2, p3);
    p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
    pointsToLine(p2, p, l2);
    areIntersect(l1, l2, ctr);
    return 1;
```

```
}
// radius of Circle Outside of a Triangle
double rCircumCircle(double ab, double bc, double ca) {     // ab, ac, ad are sides of triangle
    return ab * bc * ca / (4.0 * TriangleArea(ab, bc, ca));
}
point CircumCircleCenter(point a, point b, point c, double &r) {    // returns certer and radius of circumcircle
    double ab = dist(a, b);
    double bc = dist(b, c);
    double ca = dist(c, a);
    r = rCircumCircle(ab, bc, ca);
    if(Equal(r, ab))    return point((a.x+b.x)/2, (a.y+b.y)/2);
    if(Equal(r, bc))    return point((b.x+c.x)/2, (b.y+c.y)/2);
    if(Equal(r, ca))    return point((c.x+a.x)/2, (c.y+a.y)/2);
    line AB, BC;
    pointsToLine(a, b, AB);
    pointsToLine(b, c, BC);
    line perpenAB = perpendicularLine(AB, point((a.x+b.x)/2, (a.y+b.y)/2));
    line perpenBC = perpendicularLine(BC, point((b.x+c.x)/2, (b.y+c.y)/2));
    point center;
    areIntersect(perpenAB, perpenBC, center);
    return center;
}

// --------- Trapizoid ---------
double TrapiziodArea(double a, double b, double c, double d) {      // a and c are parallel
    double BASE = fabs(a-c);
    double AREA = TriangleArea(d, b, BASE);
    double h = (AREA*2)/BASE;
    return ((a+c)/2)*h;
}

// Hashing
// p = 31, 51
// MOD = 1e9+9, 1e7+7
const ll p = 31;
const ll mod1 = 1e9+9, mod2 = 1e9+7;

// Returns Single Hash Val
ll hash(char *s,  int len, ll mod = 1e9+9) {
    int p = 31;
    ll hashVal = 0;
    ll pPow = 1;
    for(int i = 0; i < len; ++i) {
        hashVal = (hashVal + (s[i] - 'a' + 1) * pPow)%mod;
        pPow = (pPow *p)%mod;
    }
    return hashVal;
}
vl Hash(char *s, int len) {
    ll hashVal = 0;
    vector<ll>v;
    for(int i = 0; i < len; ++i) {
        hashVal = (hashVal + (s[i] - 'a' + 1)* Power[i])%mod;
        v.push_back(hashVal);
    }
    return v;
```

```
}
bool MATCH(pll a, pll b) {
    while(a.fi <= a.se) {
        if(s1[a.fi] != s2[b.fi])
            return 0;
        a.fi++, b.fi++;
    }
    return 1;
}
void PowerGen(int n) {
    Power.resize(n+1);
    Power[0] = 1;
    for(int i = 1; i < n; ++i)
        Power[i] = (Power[i-1] * p)%mod;
}
ll SubHash(vl &Hash, ll l, ll r, ll LIM) {
    ll H;
    H = (Hash[r] - (l-1 >= 0 ? Hash[l-1]:0) + mod)%mod;
    H = (H * Power[LIM-l])%mod;
    return H;
}

// --------------- DOUBLE HASH GENERATORS ---------------
// Generates Hash of entire string without PowerGen func
vector<pair<ll, ll> > doubleHash(char *s, int len, ll mod1 = 1e9+7, ll mod2 = 1e9+9) {
    ll hashVal1 = 0, hashVal2 = 0, pPow1 = 1, pPow2 = 1;
    vector<pair<ll, ll> >v;
    for(int i = 0; i < len; ++i) {
        hashVal1 = (hashVal1 + (s[i] - 'a' + 1)* pPow1)%mod1;
        hashVal2 = (hashVal2 + (s[i] - 'a' + 1)* pPow2)%mod2;
        pPow1 = (pPow1 * p)%mod1;
        pPow2 = (pPow2 * p)%mod2;
        v.push_back({hashVal1, hashVal2});
    }
    return v;
}
void PowerGen(int n) {
    Power.resize(n+1);
    Power[0] = {1, 1};
    for(int i = 1; i < n; ++i) {
        Power[i].first = (Power[i-1].first * p)%mod1;
        Power[i].second = (Power[i-1].second * p)%mod2;
}}
vll doubleHash(char *s, int len) {      // Returns Double Hash vector for a full string
    ll hashVal1 = 0, hashVal2 = 0;
    vector<pll>v;
    for(int i = 0; i < len; ++i) {
        hashVal1 = (hashVal1 + (s[i] - 'a' + 1)* Power[i].fi)%mod1;
        hashVal2 = (hashVal2 + (s[i] - 'a' + 1)* Power[i].se)%mod2;
        v.push_back({hashVal1, hashVal2});
    }
    return v;
}
pll SubHash(vll &Hash, ll l, ll r, ll LIM) {    // Produce SubString Hash
    pll H;
    H.fi = (Hash[r].fi - (l-1 >= 0 ? Hash[l-1].fi:0) + mod1)%mod1;
    H.se = (Hash[r].se - (l-1 >= 0 ? Hash[l-1].se:0) + mod2)%mod2;
    H.fi = (H.fi * Power[LIM-l].fi)%mod1;
    H.se = (H.se * Power[LIM-1].se)%mod2;
    return H;
}
// Returns True if the Hashval of length len exists in subrange [l, r] of Hash vector
bool MatchSubStr(int l, int r, vector<pll>&Hash, pll HashVal, int len) {
    for(int Start = l, End = l+len-1; End <= r; ++End, ++Start) {
        pll pattHash, strHash;
        pattHash.first = (HashVal.first*Power[Start].first)%mod1;
        pattHash.second = (HashVal.second*Power[Start].second)%mod2;
        strHash.first = (Hash[End].first - (Start == 0 ? 0:Hash[Start-1].first) + 
mod1)%mod1;
        strHash.second = (Hash[End].second - (Start == 0 ? 0:Hash[Start-1].second) + 
mod2)%mod2;
        if(strHash == pattHash) return 1;
    }
    return 0;
}

// Heavy Light Decomopse + Segment Tree
// Tree node value update, Tree node distance

int parent[MAX], level[MAX], nextNode[MAX], chain[MAX], num[MAX], val[MAX], numToNode[MAX], 
top[MAX], ChainSize[MAX], mx[MAX];
int ChainNo = 1, all = 1, n;
vi G[MAX];
void dfs(int u, int Parent) {
    parent[u] = Parent;             // Parent of u
    ChainSize[u] = 1;               // Number of child (initially the size is 1, contains 
only 1 node. itself) (resued array in hld)
    for(int i = 0; i < SIZE(G[u]); ++i) {
        int v = G[u][i];
        if(v == Parent)                         // if the connected node is parent, skip
            continue;
        level[v] = level[u]+1;                  // level of the child node is : level of 
parent node + 1
        dfs(v, u);
        ChainSize[u] += ChainSize[v];           // Modify this line if max Chain is needed
        if(nextNode[u] == -1 || ChainSize[v] > ChainSize[nextNode[u]])
            nextNode[u] = v;            // next selected node of u (select the node which 
has more child, (HEAVY))
}}
void hld(int u, int Parent) {
    chain[u] = ChainNo;                 // Chain Number
    num[u] = all++;                     // Numbering all nodes
    if(ChainSize[ChainNo] == 0)         // if this is the first node
        top[ChainNo] = u;               // mark this as the root node of the n'th chain
    ChainSize[ChainNo]++;
    if(nextNode[u] != -1)               // if this node has a child, go to it
        hld(nextNode[u], u);            // the next node is included in the chain
    for(int i = 0; i < SIZE(G[u]); ++i) {
        int v = G[u][i];                // if this node is parent node or, this node is 
already included in the chain, skip
        if(v == Parent || v == nextNode[u]) continue;
        ChainNo++;                              // this is a new (light) chain, so increment the 
chain no. counter
        hld(v, u);
}}
int GetSum(int u, int v) {
    int res = 0;
    while(chain[u] != chain[v]) {                           // While two nodes are not in
same chain
        if(level[top[chain[u]]] < level[top[chain[v]]])     // u is the chain which's
topmost node is deeper
            swap(u, v);
        int start = top[chain[u]];
        res += query(1, 1, n, num[start], num[u]);          // Run query in u node's chain
        u = parent[start];                                  // go to the upper chain of u
    }
    if(num[u] > num[v]) swap(u, v);
    res += query(1, 1, n, num[u], num[v]);
    return res;
}
void updateNodeVal(int u, int val) {
    update(1, 1, n, num[u], val);                   // Updating the value of chain
}
void numToNodeConv(int n) {
    for(int i = 1; i <= n; ++i) numToNode[num[i]] = i;
}
int main() {
    memset(nextNode, -1, sizeof nextNode);
    ChainNo = 1, all = 1;
    dfs(1, 1);
    memset(ChainSize, 0, sizeof ChainSize);     // array reused in hld
    hld(1, 1);
    numToNodeConv(n);
    init(1, 1, n);
}

// Interval Sum
// Complexity: query*log(query)
// http://codeforces.com/contest/915/problem/E

struct Interval {
    set<pair<ll, ll> >Set;                      // Contains Segment Endpoints {r, l}
    map<pair<ll, ll>, ll>Val;                   // Contains Segment Values    {l, r} = k
    int TOTlen;                                 // Contains Total Segment Covered Length
    void init(int sz = -1) {
        Set.clear(), Val.clear(), TOTlen = 0;
        if(sz > 0)                              // Will be initialized if size declared (NOT
needed)
            Set.insert(make_pair(sz, 1)), Val[make_pair(1, sz)] = 0;
    }
    void Insert(ll l, ll r, ll val) {
        set<pair<ll, ll> > :: iterator it = Set.lower_bound({l, 0LL});
        while(it != Set.end() && it->second <= r) {
            ll segL = it->second, segR = it->first;             // Overlapped segment
            Set.erase(it++);                                    // Erase and point to the
next segment
            ll L = max(segL, l), R = min(segR, r);              // Erased segment's partial
L and R
            TOTlen -= R-L+1;
            if(segL < l) {
                Set.insert({l-1, segL});
                Val[{segL, l-1}] = Val[{segL, segR}];
            }
            if(segR > r) {
                Set.insert({segR, r+1});
                Val[{r+1, segR}] = Val[{segL, segR}];
            }
            Val.erase({segL, segR});
        }
        TOTlen += r-l+1;
        Set.insert(make_pair(r, l));
        Val[make_pair(l, r)] = val;
    }
    ll getSum(ll l, ll r) {
        ll sum = 0;
        set<pair<ll, ll> > :: iterator it = Set.lower_bound({l, 0LL});
        while(it != Set.end() && it->second <= r) {
            ll segL = it->second, segR = it->first;                 // Overlapped segment
            ll V = Val[{segL, segR}];
            sum += (segR-segL+1) * V;
            if(segL < l) sum -= (l-segL)*V;
            if(segR > r) sum -= (segR-r)*V;
            ++it;
        }
        return sum;
}};
vector<ll> CountInterval(int n) // returns number of overlaps of all inclusive points
    vector<pair<ll, int> >v;    // segments start/end and marker (segments are l - r
inclusive)
    vector<ll>ret(n+1);         // returns : ret[number_of_overlaps] =
total_number_of_points
    ll l, r;
    while(n--) {
        cin >> l >> r;                                  // input
        v.push_back({l, 1}), v.push_back({r+1, -1});    // r+1 as l - r is segment boundary
    }
    sort(v.begin(), v.end());
    for(int i = 0, cnt = v[0].second; i < (int)v.size(); ++i, cnt += v[i].second)   // cnt
contains the overlaps
        if(i+1 < (int)v.size() and v[i].first != v[i+1].first)          // v[i].first ==
v[i+1].first then
            ret[cnt] += v[i+1].first - v[i].first   // there may exist more points at front,
so take them first
    return ret;
}

// Knuth Morris Pratt
// Complexity : O(String + Token)

//------------------------Genuine PrefixTable (Prefix-Suffix Length)------------
// Some Tricky Cases:   aaaaaa : 0 1 2 3 4 5        aaaabaa : 0 1 2 3 0 1 2     abcdabcd : 0
0 0 0 1 2 3 4
void prefixTable(int n, char pat[], int table[]) {
    int len = 0, i = 1;                     // length of the previous longest prefix suffix
    table[0] = 0;                           // table[0] is always 0
    while (i < n) {
        if pat[i] == pat[len]) {
            len++;
            table[i] = len;
            i++;
        }
        else {                                                  // pat[i] != pat[len]
```

```
            if(len != 0)    len = table[len-1];                     // find previous match
            else            table[i] = 0, i++;                      // if (len == 0) and 
mismatch
}}}                                                                 // set table[i] = 0, and 
go to next index

void KMP(int strLen, int patLen, char str[], char pat[], int table[]) {
    int i = 0, j = 0;                           // i : string index, j : pattern index
    while (i < N) {
        if(str[i] == pat[j]) i++, j++;
        if(j == M) {
            printf("Found pattern at index %d n", i-j);
            j = table[j-1];                     // Match found, try for next match
        }
        else if(i < strLen && str[i] != pat[j]) {       // Match not found
            if(j != 0)  j = table[j-1];                 // if j != 0, then go to the prev 
match index
            else        i = i+1;                        // if j == 0, then we need to go to 
next index of str
}}}

// --------------- 2D KMP ---------------

unordered_map<string, int>patt;             // Clear after each Kmp2D call
int flag = 0;                               // Set to zero before calling PrefixTable
// r : Pattern row, c : Pattern column      // table : prefix table (1D array)
// s : Pattern String (C++ string)          // Followed Felix-Halim KMP
vector<int> PrefixTable2D(int r, int c, int table[], string s[]) {
    vector<int>Row;                     // Contains Row mapped string index
    for(int i = 0; i < r; ++i) {
        if(patt.find(s[i]) == patt.end()) {
            patt[s[i]] = ++flag;
            Row.push_back(flag);
        }
        else Row.push_back(patt[s[i]]);
    }
    table[0] = -1;
    int i = 0, j = -1;
    while(i < r) {
        while(j >= 0 && Row[i] != Row[j])
            j = table[j];
        ++i, ++j;
        table[i] = j;
    }
    return Row;       // Returns Hashed index of each row in pattern string
}

// StrR StrC : String Row and Column       // PattR PattC : Pattern row and column
// Str : String (C++ String)               // Patt : Pattern (C++ String)      // table : 
Prefix table of pattern (1D array)
vector<pair<int, int> > Kmp2D(int StrR, int StrC, int PattR, int PattC, string Str[], string 
Patt[], int table[]) {
    int mat[StrR][StrC];
    int limC = StrC - PattC;
    vector<int>PattRow = PrefixTable2D(PattR, PattC, table, Patt);
    for(int i = 0; i < StrR; ++i)
        for(int j = 0; j <= limC; ++j) {
            string tmp = Str[i].substr(j, PattC);
            if(patt.find(tmp) == patt.end()) {          // Generating String Mapped using same mapping values
                patt[tmp] = ++flag;                     // Stored in matrix
                mat[i][j] = flag;
            }
            else mat[i][j] = patt[tmp];
        }
    vector<pair<int, int> >match;                       // This will contain the starting Row & Column of matched string
    for(int c = 0; c <= limC; ++c) {                    // Scan columnwise
        int i = 0, j = 0;
        while(i < StrR) {
            while(j >= 0 && mat[i][c] != PattRow[j])
                j = table[j];
            ++i, ++j;
            if(j == PattR) match.push_back(make_pair(i-j,c));
    }}
    return match;
}

// LCA
// Least Common Ancestor with sparse table

vl G[MAX], W[MAX];
int level[MAX], parent[MAX], sparse[MAX][20];
ll dist[MAX], DIST[MAX][20];

void dfs(int u, int par, int lvl, ll d) {           // Tracks distance as well (From root 1 to all nodes)
    level[u] = lvl;                                 // parent[] and level[] is necessary
    parent[u] = par;
    dist[u] = d;                                    // remove distance if not needed
    for(int i = 0; i < (int)G[u].size(); ++i)
        if(parent[u] != G[u][i])
            dfs(G[u][i], u, lvl+1, d+W[u][i]);
}

void LCAinit(int V) {
    memset(parent, -1, sizeof parent);
    dfs(0, -1, 0);                                 // DFS first
    memset(sparse, -1, sizeof sparse);              // Main initialization of sparse table LCA starts here
    for(int u = 1; u <= V; ++u)                     // node u's 2^0 parent
        sparse[u][0] = parent[u];
    for(int p = 1, v; (1LL<<p) <= V; ++p)
        for(int u = 1; u <= V; ++u)
            if((v = sparse[u][p-1]) != -1)      // node u's 2^x parent = parent of node v's 2^(x-1) [ where node v : (node u's 2^(x-1) parent) ]
                sparse[u][p] = sparse[v][p-1];
}

int LCA(int u, int v) {
    if(level[u] > level[v]) swap(u, v);         // v is deeper
    int p = ceil(log2(level[v]));

    for(int i = p ; i >= 0; --i)                // Pull up v to same level as u
        if(level[v] - (1LL<<i) >= level[u])
            v = sparse[v][i];
    if(u == v) return u;                // if u WAS the parent

    for(int i = p; i >= 0; --i)                                     // Pull up u and v together while LCA not found
        if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i])      // -1 check is if 2^i is out of calculated range
            u = sparse[u][i], v = sparse[v][i];
    return parent[u];
}

// --------------------------- LCA WITH DISTANCE ---------------------------
void distDP(int V) {                    // initialiser for LCA_with_DIST, call after LCAinit()
    for(int u = 1; u <= V; ++u)         // NOTE : DIST[u][0] = weight of node u
        DIST[u][0] = W[u];              // Where W[u] = weight of node u
    for(int p = 1; (1<<p)<=V; ++p)
        for(int u = 1; u <=V; ++u) {
            int v = sparse[u][p-1];
            if(v == -1) continue;
            DIST[u][p] += DIST[u][p-1] + DIST[v][p-1];
}}

int LCA_with_DIST(int u, int v, long long &w) {     // w retuns distance from u -> v
    w = 0;
    if(level[u] > level[v]) swap(u, v);             // v is deeper
    int p = ceil(log2(level[v]));
    for(int i = p ; i >= 0; --i)                    // Pull up v to same level as u
        if(level[v] - (1LL<<i) >= level[u]) {
            w += DIST[v][i];
            v = sparse[v][i];
        }
    if(u == v) {                                    // if u WAS the parent
        w += DIST[v][0];
        return u;
    }
    for(int i = p; i >= 0; --i)                     // Pull up u and v together while LCA not found
        if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i])      // -1 check is if 2^i is out of calculated range
            u = sparse[u][i], v = sparse[v][i];
    w += DIST[v][0];
    w += DIST[u][0];
    w += DIST[sparse[v][0]][0];
    return parent[u];
}

ll Distance(int u, int v) {
    int lca = LCA(u, v);
    return dist[v] + dist[u] - 2*dist[lca];
}

// --------------------------- LCA WITH Sparse Table Vector ---------------------------
// DFS and LCA INIT is same
void MERGE(vector<int>&u, vector<int>&v) {          // Do what is to be done to merge
    for(auto it : v) u.push_back(it);               // here taking lowest 10 values
    sort(u.begin(), u.end());
    while((int)u.size() > 10)
        u.pop_back();
```

```
}

vector<int> W[MAX][20];             // W[u][0] will contain initial weight/weights at node u
vector<int> LCA(int u, int v) {
    vector<int> T;
    if(level[u] > level[v]) swap(u, v);     // v is deeper
    int p = ceil(log2(level[v]));
    for(int i = p ; i >= 0; --i)            // Pull up v to same level as u
        if(level[v] - (1LL<<i) >= level[u]) {
            MERGE(T, W[v][i]);
            v = sparse[v][i];
        }
    if(u == v) {                                // if u WAS the parent
        MERGE(T, W[u][0]);
        return T;
    }
    for(int i = p; i >= 0; --i)                                 // Pull up u and v together while LCA not found
        if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i]) {        // -1 check is if 2^i is out of calculated range
            MERGE(T, W[u][i]);
            MERGE(T, W[v][i]);
            u = sparse[u][i], v = sparse[v][i];
        }
    MERGE(T, W[u][0]);              // As W[x][0] denoted the x nodes weight
    MERGE(T, W[v][0]);              // every sparse node must be calculated
    MERGE(T, W[sparse[v][0]][0]);   // we can also calculate summation of distance like this
    return T;
}

// Longest Increasing/Decrasing Sequence
// Complexity : nLog_n

// -----------------------Non Printable Version--------------------

int main() {
    // vector v contains the sequence
    for(auto it : v) {                                      // Use -it for decrasing sequences
        auto pIT = upper_bound(LIS.begin(), LIS.end(), it);     // Longest Non-Decreasing Sequence
        if(pIT == LIS.end())                                // For Longest Increasing Sequence use lower_bound
            LIS.push_back(it);
        else
            *pIT = it;
    }
    return 0;
}

// -------------------------Printable Version------------------------
// DP + BinarySearch (nLog_n)
// {1, 1, 9, 3, 8, 11, 4, 5, 6, 6, 4, 19, 7, 1, 7}
// Incrasing     : 1, 3, 4, 5, 6, 7
// NonDecreasing : 1, 1, 3, 4, 5, 6, 6, 7, 7

void findLIS(vector<int> &v, vector<int> &idx) {    // v is the input values and idx vector contains index of the LIS values
```

```
    if(v.empty()) return;
    vector<int> dp(v.size());               // The memoization part, remembers what index is the previous index if any value is inserted or modified
    idx.push_back(0);                       // Carrys index of values
    int l, r;

    for(int i = 1; i < (int)v.size(); i++) {
        if(v[idx.back()] <= v[i]) {         // **Replace < with <= if non-decreasing subsequence required
            dp[i] = idx.back();             // If next element v[i] is greater than last element of
            idx.push_back(i);               // current longest subsequence v[idx.back()], just push it at back of "idx" and continue
            continue;
        }
        // Binary search to find the smallest element referenced by idx which is just bigger than v[i] (UpperBound(v[i]))
        // Note : Binary search is performed on idx (and not v)
        for(l = 0, r = idx.size()-1; l < r; ) {
            int mid = (l+r)/2;
            if(v[idx[mid]] <= v[i])  l = mid+1;      // **Replace < with <= if non-decreasing subsequence required
            else                     r = mid;
        }
        if(v[i] < v[idx[l]]) {                      // Update idx if new value is smaller then previously referenced value
            if(l > 0) dp[i] = idx[l-1];
            idx[l] = i;
        }
    }
    for(l = idx.size(), r = idx.back(); l--; r = dp[r])
        idx[l] = r;
}

// Math Formulas

// Find the number of b for which [b1, b2] | [a1, a2]
int FindDivisorInRange(int a1, int a2, int b1, int b2) {
    int a = abs(a1 - a2);
    int b = abs(b1 - b2);
    int gcd = __gcd(a, b);
    return 1 + gcd;
}

// Find how many integers from range m to n are divisible by a or b
int rangeDivisor(int m, int n, int a, int b) {
    int lcm = LCM(a, b);
    int a_divisor = n / a - (m - 1) / a;
    int b_divisor = n / b - (m - 1) / b;
    int common_divisor = n / lcm - (m - 1) / lcm;
    int ans = a_divisor + b_divisor - common_divisor;
    return ans;
}

ll CSOD(ll n) {                             // Cumulative Sum of Divisors in sqrt(n)
    ll ans = 0;
    for(ll i = 2; i * i <= n; ++i) {
        ll j = n / i;
        ans += (i + j) * (j - i + 1) / 2;
        ans += i * (j - i);
    }
    return ans;
}

int CountDivisible(int a, int b, int n) {   // Returns the number of divisible value in range [a, b] by n (NOT TESTED)
    if(a > b) swap(a, b);
    a += n - a%n;
    b -= b%n;
    if(a > b) return 0;
    return ceil((b-a+1)/(double)n);
}

int FactorialCount(int n, int p = 5) {      // Returns how many value of p is present in n!
    int ret = 0, r = p;                     // returns number of trailing zero of n! if p = 5
    while(n/r != 0) {
        ret += n/r;
        r *= p;
    }
    return ret;
}

int TrailingZero(int n, int p = 1) {        // Returns Trailing Zero of n^p
    int cnt = 0;                            // Trailing Zero for any number : min(count_2_as_prime_factor, count_5_as_prime_factor)
    while(n%5 == 0 && n%2 == 0)
        n /= 5, n /= 2, ++cnt;
    return cnt*p;
}

int BirthdayParadox(int days, int targetPercent = 50) {         // Returns Number of people required so that probability is >= target
    int people = 0;                                             // Formula : 1 - (365/365) * (364/365) * (363/365) * .....
    double percent = targetPercent/100.0, gotPercent = 1;
    for( ; gotPercent > percent; ++people)
        gotPercent *= (days-people-1)/(double)days;
    return people;
}

/* Euler’s Totient function Φ(n) for an input n is count of numbers in {1, 2, 3, …, n}
 * that are relatively prime to n, i.e., the numbers whose GCD (Greatest Common Divisor) with n is 1.
 * Phi(4) :  GCD(1, 4) = 1,  GCD(3, 4)
 * so, Phi(4) = 2
 */

int Phi(int n) {                        // Computes phi of n
    int result = n;
    for(int p=2; p*p<=n; ++p) {         // Consider all prime factors of n and subtract their multiples from result
        if(n % p == 0) {                // p is a prime factor of n
            while (n % p == 0)          // eleminate all p factors from n
                n /= p;
            result -= result / p;
    }}
    if(n > 1)                           // if n is still greater than 1, then it is also a prime
        result -= result / n;
    return result;
}

long long phi[MAX];
void computeTotient(int n) {            // Computes phi or Euler Phi 1 to n
    for (int i=1; i<=n; i++)            // Initialize
        phi[i] = i;
    for (int p=2; p<=n; p++) {          // Computation
        if (phi[p] == p) {              // if phi is not computed
            phi[p] = p-1;               // p is prime and phi(prime) = prime-1;
            for (int i = 2*p; i<=n; i += p) {       // Sieve like implementation
               phi[i] = (phi[i]/p) * (p-1);         // Add contribution of p to its multiple i by multiplying with (1 - 1/p)
}}}}

// Combination
// Complexity O(k)
long long C(int n, int k) {
    long long c = 1;
    if(k > n - k)
        k = n-k;
    for(int i = 0; i < k; i++) {
        c *= (n-i);
        c /= (i+1);
    }
    return c;
}

ll fa[MAX], fainv[MAX];                                 // fa and fainv must be in global
ll C(ll n, ll r) {                                      // Usable if MOD value is present
    if(fa[0] == 0) {                                    // Auto initialize
        fa[0] = 1, fainv[0] = powerMOD(1, MOD-2);
        for(int i = 1; i < MAX; ++i) {
            fa[i] = (fa[i-1]*i) % MOD;                  // Constant MOD
            fainv[i] = powerMOD(fa[i], MOD-2);
    }}
    if(n < 0 || r < 0 || n-r < 0) return 0;             // Exceptional Cases
    return ((fa[n] * fainv[r])%MOD * fainv[n-r])%MOD;
}

ll Catalan(int n) {    // Cat(n) = C(2*n, n)/(n+1);
    ll c = C(2*n, n);
    return c/(n+1);
}

// Building Pascle C(n, r)
ll p[MAX][MAX];
void buildPascle() {                                    // This Contains values of nCr : p[n][r]
    p[0][0] = 1;
    p[1][0] = p[1][1] = 1;
    for(int i = 2; i <= 50; i++)
        for(int j = 0; j <= i; j++) {
            if(j == 0 || j == i)
                p[i][j] = 1;
            else
                p[i][j] = p[i-1][j-1] + p[i-1][j];
}}

ll C(int n, int r) {
    if (r<0 || r>n) return 0;
    return p[n][r];
}

// STARS AND BARS THEOREM / Ball and Urn theorem
// If We have to Make x1+x2+x3+x4 = 12
// Then, the solution can be expressed as : {*|*****|****|**} = {1+5+4+2}, {|*****|***|****} = {0+5+3+4}
// The summation is presented as total value, and the stars represanted as 1, we use bars to seperate values
// Number of ways we can produce the summation n, with k unknowns : C(n+k-1, n) = C(n+k-1, k-1)

// If numbers have lower limits, like x1 >= 3, x2 >= 2, x3 >= 1, x4 >= 1   (Let, the lower limits be l[i])
// Then the solution is : C(n-l1-l2-l3-l4+k-1, k-1)

// Ball & Urn : how many ways you can put 1 to n number in k sized array so that ther are non decreasing?

ll StarsAndBars(vector<int> &l, int n, int k) {
    if(!l.empty()) for(int i = 0; i < k; ++i) n -= l[i];        // If l is empty, then there is no lower limit
    return C(n+k-1, k-1);
}

// If numbers have both boudaries l1 <= x1 <= r1, l2 <= x2 <= r2, and x1+x2 = N
// then we can reduce the form to x1+x2 = N-l1-l2 and then x only gets upper limit x1 <= r1-l1+1, x2 <= r2-l2+1
// let r1-l1+1 be new l1, and r2-l2+1 be new l2, so x1 <= l1 and x2 <= l2, this limit is the opposite of basic Stars
// and Bars theorem, according to Principle of Inclusion Exclution, this answer can be found as
// Answer = C(n+k-1, k-1) - C(n-l1+k-1, k-1) - C(n-l2-k-1, k-1) + C(n-l1-l2+k-1, k-1) ......

ll StarsAndBarsInRange(ll l[], ll r[], ll n, ll k) {
    ll d[k+10], p[(1<<k) + 10];
    for(int i = 0; i < k; ++i) {
        d[i] = r[i] - l[i] + 1;
        n -= l[i];
    }
    ll ret = C(n+k-1, k-1); p[0] = 0;
    for(int i = 0; i < k; ++i)                                  // Optimized Complexity : 2^n
        for(int mask = (1<<i); mask < (1 << (i+1)); ++mask) {
            p[mask] = p[mask ^ (1<<i)] + d[i];
            ret += C(n-p[mask]+k-1, k-1) * (__builtin_popcount(mask)&1 ? -1:1);
            ret %= MOD;
        }
    return (ret+MOD)%MOD;
}

```

```
vll GetSameMOD(vector<ll>&v) {      // Given an array v, find values k (k > 1), for which v[0]%k = v[1]%k ... = v[n]%k
    ll gcd;                         // If a number K, leaves the same remainder with 2 numbers, then it must divide their difference.
    sort(v.begin(), v.end());       // Find all numbers K which divide all the consecutive differences of all elements in the array.

    for(int i = 0; i+1 < (int)v.size(); ++i) {      // And it we will take the GCD of all consecutive differences
        if(i == 0)  gcd = v[i+1] - v[i];
        else        gcd = __gcd(gcd, v[i+1] - v[i]);
    }
    vector<ll> ret = Divisors(gcd);         // GCD is the maximum value of k
    ret.push_back(gcd);                     // All other values are the divisors of k
    sort(ret.begin(), ret.end());           // NOTE : 1 is not added in the answer
    return ret;
}

ll CountZerosInRangeZeroTo(string n) {              // Returns number of zeros from 0 to n
    ll x = 0, fx = 0, gx = 0;
    for(int i = 0; i < (int)n.size(); ++i){
        ll y = n[i] - '0';
        fx = 10LL * fx + x - gx * (9LL - y);        // Our formula
        //fx += MOD;                                // If ans is to be returned in moded value
        //fx %= MOD;
        x = 10LL * x + y;                           // Now calculate the new x and g(x)
        //x %= MOD;
        if(y == 0LL) gx++;
    }
    return fx+1;
}

ll NumOfSameValueInCombination(int n, int r) {          // Returns number of same value in a set of nCr combination
    if(n < r) return 0;
    return C(n-1, r-1);
}

int cnt[MAX];                                           // cnt[x] : how many times x occures in input
vector<int> genGCD(int mx) {                            // Counts how many number are there of gcd x
    vector<int>sameGCD(mx+1);                           // input the MAXIMUM value
    for(int gcd = mx; gcd >= 2; --gcd) {                // Complexity : mx log_mx
        int gcdCNT = cnt[gcd];
        for(int mul = gcd+gcd; mul <= mx; mul += gcd)
            gcdCNT += cnt[mul];
        sameGCD[gcd] = gcdCNT;
    }
    return sameGCD;
}

// Multinomial : nC(k1,k2,k3,..km)    is such that k1+k2+k3+....km = n and ki == kj and ki != kj both can be possible.
// Here Multinomial can be described as : nC(k1, k2, .. km) = nCk1 * (n-k1)Ck2 * (n-k1-k2)Ck3 * ..... (n-k1-k2-...)Ckm
// Let (a+b+c)^3 = a^3 + b^3 + c^3 + 3a^2b + 3b^c + 3b^2a + 3b^2c + 3c^2a + 3c^2b + 6abc
// The coefficient can be retrieved as : 6abc = 3C(1, 1, 1) = 6     3b^2c = 3C(0, 2, 1) = 3
// In general terms it tells how many ways we can place k1, k2, k3, k4 people in 3 unique teams such that k1+k2+k3
// NOTE: if k1=k2=k3 = 2 and n = 6, and players numberd from 1 to 6, then 1,2|3,4|5,6 and 3,4|1,2|5,6 are considered to be different

ll fa[MAX] = {0};                              // fa and fainv must be in global
ll Multinomial(ll N, vector<ll>& K) {          // K contains all k1, k2, k3,  if k1=k2=k3, then just push k1 once!!
    if(fa[0] == 0) {                           // Auto initialize
        fa[0] = 1; //fainv[0] = powerMOD(1, MOD-2);
        for(int i = 1; i < MAX; ++i) {
            fa[i] = (fa[i-1]*i) % MOD;                    // Constant MOD
            //fainv[i] = powerMOD(fa[i], MOD-2);          // Use factorial inverse if required
    }}
    ll k = 1;
    if((int)K.size() == 1)  k = powerMOD(fa[K[0]], N/K[0]);         // k1 = k2 = .. = km, so k occurs N/K time
    else for(auto it : K)   k = (k*fa[it])%MOD;
    return (fa[N]*powerMOD(k, MOD-2))%MOD;                          // Inverse mod
}

ll NumOfWaysToPlace(ll N, ll K) {              // Number of ways to make N/K teams from N people so that each team contais K people
    vector<ll>v;                               // If N = 6, then 1,2|3,4|5,6  and  3,4|1,2|5,6 is considered same
    v.push_back(K);
    return (Multinomial(N, v)*powerMOD(fa[N/K], MOD-2))%MOD;    // divide by k!, as 1,2|3,4|5,6  and  3,4|1,2|5,6 is considered same
}

ull partial_derangement(int n, int r) {         // Finds out how many ways we can place n numbers where r of them are not in their initial place
    ull ans = f[n];                             // Factorial of n!
    for(int i = 1; i <= r; ++i) {     // Formula: n! - C(n, 1)*(n-1)! + C(n, 2)*(n-2)! ..... + (-1)^r * C(n,r)*(n-r)!
        if(i & 1) ans = (ans%MOD - (C(r, i) * f[n-i])%MOD)%MOD; // Here C(r, i) is because we only have to choose from r elements, not n elements
        else      ans = (ans%MOD + (C(r, i) *f[n-i])%MOD)%MOD;
        ans = (ans + MOD)%MOD;
    }
    return ans%MOD;
}

struct matrix {
    matrix() { memset(mat, 0, sizeof(mat)); }
    long long mat[MAXN][MAXN];
};
matrix mul(matrix a, matrix b, int p, int q, int r) {       // O(n^3) :: r1, c1, c2  [c1 = r2]
    matrix ans;
    for(int i = 0; i < p; ++i)
        for(int j = 0; j < r; ++j) {
            ans.mat[i][j] = 0;
            for(int k = 0; k < q; ++k)
                ans.mat[i][j] = (ans.mat[i][j]%MOD + (a.mat[i][k]%MOD * b.mat[k][j]%MOD)%MOD)%MOD;
        }
    return ans;
}
matrix matPow(matrix base, ll p, int s) {               // O(logN), s : size of square matrix
    if(p == 1) return base;
    if(p & 1)  return mul(base, matPow(base, p-1, s), s, s, s);
    matrix tmp = matPow(base, p/2, s);
    return mul(tmp, tmp, s, s, s);
}

// MaxFlow
// Ford-Fulkerson
// Complexity: O(VE^2)
// Graph Type : Directed/Undirected

const int MAX = 120;
vector<int>edge[MAX];
int V, E, rG[MAX][MAX], parent[MAX];

bool bfs(int s, int d) {                // augment path : source, destination
    memset(parent, -1, sizeof parent);
    queue<int>q;
    q.push(s);
    while(!q.empty()) {
        int u = q.front();
        q.pop();
        for(auto v : edge[u])
            if(parent[v] == -1 && rG[u][v] > 0) {
                parent[v] = u;
                if(v == d) return 1;
                q.push(v);
    }}
    return 0;
}

int maxFlow(int s, int d) {             // source, destination
    int max_flow = 0;
    while((bfs(s, d))) {
        int flow = INT_MAX;
        for(int v = d; v != s; v = parent[v]) {
            int u = parent[v];
            flow = min(flow, rG[u][v]);
        }
        for(int v = d; v != s; v = parent[v]) {
            int u = parent[v];
            rG[u][v] -= flow;
            rG[v][u] += flow;
        }
        max_flow += flow;
    }
    return max_flow;
}

int main() {
    int u, v, w, source, destination, Case = 1;
    map<pair<int, int>, bool>Map;
    while(scanf("%d", &V) && V) {
        scanf("%d%d%d", &source, &destination, &E);
        memset(rG, 0, sizeof rG);
        for(int i = 0; i < E; ++i) {
            scanf("%d%d%d", &u, &v, &w);
            rG[u][v] += w;                          // edges are undirected
            rG[v][u] += w;                          // remove this line if edges are
directed
            if(Map.find({u, v}) == Map.end()) {             // same edges might occur more
than once
                edge[u].push_back(v);                       // to avoid n^2 calculation
                edge[v].push_back(u);
                Map[{u, v}] = Map[{v, u}] = 1;
        }}
        printf("Network %d\n", Case++);
        printf("The bandwidth is %d.\n\n", maxFlow(source, destination));

        for(int i = 0; i <= V; ++i) edge[i].clear();
        Map.clear();
    }
    return 0;
}

//1D Max Sum
//Algorithm : Jay Kadane
//Complexity : O(n)

int main() {
    int n;
    scanf("%d", &n);
    int A[n+1];
    for(int i = 0; i < n; i++)
        scanf("%d", &A[i]);

    //Main part of the code
    int sum = 0, ans = 0;
    for(int i = 0; i < 9; i++) {
        sum += A[i];
        ans = max(sum, ans);                //always take the larger sum
        if(sum < 0)
            sum = 0;                        //if sum is negative, reset it (greedy)
    }
    printf("1D Max Sum : %d\n", ans);
    return 0;
}

//2D Max Sum
//Algorithm : DP, Inclusion Exclusion
//Complexity : O(n^4)

int main() {
    int row_column, A[100][100];             //A square matrix
    scanf("%d", &row_column);

    for(int i = 0; i < row_column; i++)         //input of the matrix/2D array
        for(int j = 0; j < row_column; j++) {
            scanf("%d", &A[i][j]);
            if(i > 0) A[i][j] += A[i-1][j];                     //take from right
            if(j > 0) A[i][j] += A[i][j-1];                     //take from left
            if(i > 0 && j > 0) A[i][j] -= A[i-1][j-1];         //inclusion exclusion
        }

    int maxSubRect = -1e7;
    for(int i = 0; i < row_column; i++)                 //i & j are the starting coordinate of sub-rectangle
        for(int j = 0; j < row_column; j++)
            for(int k = i; k < row_column; k++)         //k & l are the finishing coordinate of sub-rectangle
                for(int l = j; l < row_column; l++) {
                    int subRect = A[k][l];
                    if(i > 0) subRect -= A[i-1][l];
                    if(j > 0) subRect -= A[k][j-1];
                    if(i > 0 && j > 0) subRect += A[i-1][j-1];         //due to inclusion exclusion
                    maxSubRect = max(subRect, maxSubRect);
                }
    printf("2D Max Sum : %d\n", maxSubRect);
    return 0;
}

// MergeSort

void MergeSort(long long arr[], int l, int mid, int r) {
    int lftArrSize = mid-l+1, rhtArrSize = r-mid, lftArr[lftArrSize+2], rhtArr[rhtArrSize+2];

    for(int i = l, j = 0; i <= mid; ++i, ++j)
        lftArr[j] = arr[i];
    for(int i = mid+1, j = 0; i <= r; ++i, ++j)
        rhtArr[j] = arr[i];

    lftArr[lftArrSize] = rhtArr[rhtArrSize] = 1e9;      // INF value in both array (Basic merge sort algo)
    int lPos = 0, rPos = 0;
    for(int i = l; i <= r; ++i) {
        if(lftArr[lPos] <= rhtArr[rPos])
            arr[i] = lftArr[lPos++];
        else {
            arr[i] = rhtArr[rPos++];
            //cnt += lftArrSize - lPos;                // Delete this line if not needed (Min Number of Swaps)
}}}

void Divide(long long arr[], int l, int r) {
    if(l == r || l > r) return;
    int mid = (l+r)>>1;
    Divide(arr, l, mid);
    Divide(arr, mid+1, r);
    MergeSort(arr, l, mid, r);
}
main() { Divide(v, 0, n-1); }

// Modular Arithmatic

// (2^10 % 5) = powMod(2, 10, 5)
long long powMod(long long N, long long P, long long M) {
    if(P==0) return 1;
    if(P%2==0) {
        long long ret = powMod(N, P/2, M)%M;
        return (ret * ret)%M;
    }
    return ((N%M) * (powMod(N, P-1, M)%M))%M;
}

ll powerMOD(ll x, ll y) {                   // Can find modular inverse by a^(MOD-2),  a and
MOD must be co-prime
    ll res = 1;
    x %= MOD;
    while(y > 0) {
        if(y&1) res = (res*x)%MOD;          // If y is odd, multiply x with result
        y = y >> 1, x = (x * x)%MOD;
    }
    return res%MOD;
}

// 2^100 = Pow(2, 100)
unsigned long long Pow(unsigned long long N, unsigned long long P) {
    if(P == 0) return 1;
    if(P % 2 == 0) {
        unsigned long long ret = Pow(N, P/2);
        return ret*ret;
    }
    return N * Pow(N, P-1);
}

// calculate A mod B, where A : 0<A<(10^100000) (or greater)
// take input as string and process with aftermod
// calculate A^B mod M, where B : 0<A<(10^100000) (or greater)
// take input as string and process with aftermod : afterMod(inputAsString, Mod-1)      due
to fermat theorem

long long afterMod(string str, ll mod) {        // input as string, as it is big, mod is the
Mod value (Mod-1 if modding an exponentiation)
    long long ans = 0;
    string :: iterator it;

    for(it = str.begin(); it != str.end(); it++)        // mod from first to last
        ans = (ans*10 + (*it -'0')) % mod;
    return ans;
}

// Exponent of Big numbers (N^P % M)
// where N and P is bigger strings (both having length 10^5)
long long bigExpo(char *N, char *P, long long M) {
    long long base = 0, ans = 1;
    for(int i = 0; N[i] != '\0'; ++i)
        base = (base*10LL + N[i] - '0')%M;

    for(int j = 0; P[j] != '\0'; ++j)
        ans = (powMod(ans, 10, M) * powMod(base, P[j]-'0', M))%M;
    return ans;
}

// Extended Euclud
// a*x + b*y = gcd(a, b)
// Given a and b calculate x and y so that a * x + b * y = d   (where gcd(a, b) | c)
// x_ans = x + (b/d)n
// y_ans = y − (a/d)n
// where n is an integer

// Solution only exists if d | c (i.e : c is divisable by d)
ll gcdExtended(ll a, ll b, ll *x, ll *y) {          // C function for extended Euclidean
Algorithm
    if (a == 0) {                                   // Base Case
        *x = 0, *y = 1;
        return b;
    }
    ll x1, y1;                                     // To store results of recursive call
    ll gcd = gcdExtended(b%a, a, &x1, &y1);
    *x = y1 - (b/a) * x1;
    *y = x1;
    return gcd;
}

ll modInverse(ll a, ll mod) {
    ll x, y;
    ll g = gcdExtended(a, mod, &x, &y);
    if(g != 1) return -1;                  // ModInverse doesnt exist
    ll res = (x%mod + mod) % mod;          // m is added to handle negative x
    return res;
}

// MO's Algo
// Complexity : Q*sqrt(N)

struct query {
    int l, r, id;
};

const int block = 320;          // For 100000
query q[MAX];
int ans[MAX];

bool cmp(query &a, query &b) {
    int block_a = a.l/block, block_b = b.l/block;
    if(block_a == block_b)
        return a.r < b.r;
    return block_a < block_b;
}

bool cmp2(query &a,query &b){                       // Faster Comparison function
    if(a.l/block !=b.l/block)   return a.l < b.l;
    if((a.l/block)&1)           return a.r < b.r;
    return a.r > b.r;
}

void add(int x) {}       // Add x'th value in range
void remove(int x) {}    // Remove x'th value from range

int main() {
    int Q;
    scanf("%d", &Q);
    for(int i = 0; i < Q; ++i) {                 // Query input
        scanf("%d%d", &q[i].l, &q[i].r);
        --q[i].l, --q[i].r, q[i].id = i;        // NOTE : value index starts from 0
    }

    sort(q, q+Q, cmp);
    int l = 0, r = -1;
    for(int i = 0; i < Q; ++i) {
        while(l > q[i].l)   add(--l);
        while(r < q[i].r)   add(++r);
        while(l < q[i].l)   remove(l++);
        while(r > q[i].r)   remove(r--);
        ans[q[i].id] =                      // Add Constraints
    }
    return 0;
}

// Directed Minimum Spanning Tree (Edmonds' algorithm)
// Complexity : O(E*V) ~ O(E + VlogV)                           [ works in O(E + VlogV) for almost all cases ]
// https://en.wikipedia.org/wiki/Edmonds%27_algorithm

struct edge {
    int u, v, w;
    edge() {}
    edge(int a,int b,int c) : u(a), v(b), w(c) {}
};

int DMST(vector<edge> &edges, int root, int V) {
    int ans = 0;
    int cur_nodes = V;
    while(1) {
        vector<int> lo(cur_nodes, INF), pi(cur_nodes, INF);     // lo[v] : contains minimum weight to go to node v              (for an edge u -> v)
                                                                // pi[v] : contains the minimum weight edge's starting node u
        for(int i = 0; i < (int)edges.size(); ++i) {
            int u = edges[i].u, v = edges[i].v, w = edges[i].w;
            if(w < lo[v] and u != v)
                lo[v] = w, pi[v] = u;
        }

        lo[root] = 0;                                           // by default the weight to go to root node is 0
        for(int i = 0; i < (int)lo.size(); ++i) {
            if(i == root) continue;
            if(lo[i] == INF) return -1;                         // if there is no way to visit a node v, then Directed MST doesn't exist
        }

        int cur_id = 0;
        vector<int> id(cur_nodes, -1), mark(cur_nodes, -1);

        for(int i = 0; i < cur_nodes; ++i) {
            ans += lo[i];                                       // adding node i's minimum weight to answer

            int u;
            for(u = i; u != root && id[u] < 0 && mark[u] != i; u = pi[u])       // marking minimum weighted path from root to node i
```

```
                mark[u] = i;

            if(u != root && id[u] < 0) {                            // Contains cycle, as a result u can not reach to i
                for(int v = pi[u]; v != u; v = pi[v])               // mark all cycle nodes with id
                    id[v] = cur_id;
                id[u] = cur_id++;                                   // ??
        }}

        if(cur_id == 0) break;                                      // there is no cycle, so all node is possibly visited
        for(int i = 0; i < cur_nodes; ++i)
            if(id[i] < 0) id[i] = cur_id++;

        for(int i = 0; i < (int)edges.size(); ++i) {
            int u = edges[i].u, v = edges[i].v;
            edges[i].u = id[u];
            edges[i].v = id[v];
            if(id[u] != id[v]) edges[i].w -= lo[v];
        }

        cur_nodes = cur_id;
        root = id[root];
    }
    return ans;                                                     // returns total cost of MST
}

// MST Kruskal + Union Find Disjoint Set (DSU)
// Complexity of MST : O(E logV)

// Let a graph be G1, and the MST of the graph is MST1
// and a graph G2, where G2 contains same edges as G1 with some new edges
// then the new MST of graph G2 will be :
// MST2 = MST(of the edges used in M1 (MST of G1) + new added edges)

set<pair<int, pair<int, int> > >Edge;                   // USED STL SET!!

int MST(int V) {
    int mstCost = 0, edge = 0;                          // If Edge list is STL vector, then sort it!
    DSU U(V+5);
    set<pair<int, pair<int, int> > > :: iterator it = Edge.begin();     // Contains {Weight, {U, V}}

    for( ; it != Edge.end() && edge < V; ++it) {
        int u = (*it).second.first;
        int v = (*it).second.second;
        int w = (*it).first;

        if(!U.isSameSet(u, v))
            ++edge, mstCost += w, U.makeUnion(u, v);
    }

    if(edge != V-1) return -1;              // Some edge is missing, so no MST found!
    return mstCost;
}
```

```

//Minimum Spanning Tree
//Prim's Algorithm
//Complexity : O(E logV)

vector<int> G[MAX], W[MAX];
priority_queue<pair<int, int> >pq;
bitset<MAX>taken;

void process(int u) {
    taken[u] = 1;
    for(int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        int w = W[u][i];
        if(!taken[v])
            pq.push(make_pair(-w, -v));
    }
}

int main() {
    taken.reset();
    process(0);    //taking 0 node as default
    int mst_cost = 0;
    while(!pq.empty()) {
        w = -pq.top().first;
        v = -pq.top().second;
        pq.pop();
        //if the node is not taken, then use this node
        //as it contains the minimum edge
        if(!taken[v])
            mst_cost += w, process(v);
    }
    printf("Prim's MST cost : %d\n", mst_cost);
    return 0;
}

// N'th Permutation

long long per[30] = {0};
long long permute(int freq[]) {
    int cnt = 0;
    for(int i = 0; i < 26; ++i)
        cnt+=freq[i];
    long long permutation = per[cnt] < 1 ? 0:per[cnt];
    for(int i = 0; i < 26; ++i)
        if(freq[i] > 1)
            permutation /= per[freq[i]];
    return permutation;
}

string NthPermutation(string str, int n) {    // string and numbet of permutation
    string s;
    int freq[30] = {0};
    for(int i = 0; i < (int)str.size(); ++i)
        freq[str[i]-'a']++;
    if(per[0] == 0) {                          // if not initialized
        per[0] = 1;
        for(int i = 1; i <= 25; ++i)
```

```
            per[i] = per[i-1]*i;
    }
    if(permute(freq) < n)
        return s;
    for(int i = 0; i < (int)str.size(); ++i) {
        for(int j = 0; j < 26; ++j) {
            if(freq[j] <= 0) continue;
            freq[j]--;
            long long P = permute(freq);
            if(P >= n) {
                s += char(j+'a');
                break;
            }
            else {
                n -= P;
                freq[j]++;
            }}}
    return s;        // Returns empty string if not possible
}

// Palindromic Tree

struct node {
    int start, end, length, edge[26], suffixEdg;
};

struct PalinTree {
    int currNode;
    string s;                           // Contains the string
    int ptr, mxLen;
    node root1, root2, tree[MAX];
    PalinTree() {
        s.clear();
        root1.length = -1;
        root1.suffixEdg = 1;
        root2.length = 0;
        root2.suffixEdg = 1;
        tree[1] = root1;
        tree[2] = root2;
        ptr = 2;
        currNode = 1;
        mxLen = 0;
    }
    void insert(int idx) {
        int tmp = currNode;
        while(1) {
            int curLength = tree[tmp].length;
            if(idx - curLength >= 1 && s[idx] == s[idx-curLength-1]) break;
            tmp = tree[tmp].suffixEdg;
        }
        if(tree[tmp].edge[s[idx]-'a'] != 0) {
            currNode = tree[tmp].edge[s[idx]-'a'];
            return;
        }
        ptr++;
        tree[tmp].edge[s[idx]-'a'] = ptr;
        tree[ptr].length = tree[tmp].length + 2;
        tree[ptr].end = idx;
        tree[ptr].start = idx - tree[ptr].length + 1;
        tmp = tree[tmp].suffixEdg;
        currNode = ptr;
        if(tree[currNode].length == 1) {
            tree[currNode].suffixEdg = 2;
            return;
        }
        while(1) {
            int curLength = tree[tmp].length;
            if(idx-curLength >= 1 && s[idx] == s[idx-curLength-1])
                break;
            tmp = tree[tmp].suffixEdg;
        }
        tree[currNode].suffixEdg = tree[tmp].edge[s[idx]-'a'];
    }
    void buildTree() {          // Builds Palindrome Tree of string s
        for(int i = 0; i < (int)s.length(); ++i)
            insert(i);
    }
    void CalMaxLen() {
        for(int i = 3; i <= ptr; ++i)
            mxLen = max(mxLen, tree[i].end - tree[i].start);
}};

int main() {
    PalinTree pt;
    cin >> pt.s;
    pt.buildTree();
    cout << "All distinct palindromic substring for " << pt.s << " : \n";
    for(int i=3; i<=pt.ptr; i++) {
    cout << i-2 << ") ";
    for(int j=pt.tree[i].start; j<=pt.tree[i].end; j++)
        cout << pt.s[j];
        cout << endl;
}}

// Policy Based Data Structures
// Source : http://codeforces.com/blog/entry/11080
//          http://codeforces.com/blog/entry/13279
// Problems:
// http://codeforces.com/blog/entry/53735
// http://codeforces.com/contest/762/problem/E

#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>    // rb_tree_tag
#include <ext/pb_ds/tree_policy.hpp>        // tree_order_statistics_node_update
#define at(X)           find_by_order(X)
#define lessThan(x)     order_of_key(X)
using namespace std;
using namespace __gnu_pbds;
template<class T> using ordered_set = tree<T, null_type, less_equal<T>, rb_tree_tag,
tree_order_statistics_node_update>;

// key, Mapped-Policy, Key Comparison Func, Underlying data Structure, Policy for updating
node interval

// Key Comparison Func : Defines how data will be stored (incleasing, decrasing order)
//                       less<int>          -   Same value occurs once & increasing
        SET
//                       less_equal<int>    -   Same value occurs one or more & increasing
       MULTISET
//                       greater<int>, greater_equal<int>
//
// if Mapped-Policy set to null_type, this works as a SET
// else works as MAP
//
// Underlying Data Structures : rb_tree_tag     -   Red Black Tree
//                              splay_tree_tag  -   Splay Tree
//                              ov_tree_tag     -   Ordered Vector Tree
//
// Policy for Updaing Node :  default    -   null_node_update
//                   c++ immplemented    -   tree_order_statistics_node_update

// Features :
// Can be used as SET/MULTISET
// lower_bound and upper bound works as expected
// insertion, deletation, clear
// container.find_by_order(x) returns x'th elements iterator
// container.order_of_key(x) returns number of values less than (or equal to) x
// auto casting works
//

int main() {
    ordered_set<int> X;
    // -------------- INSERTION --------------
    // Data are indexed in increasing order & can occur only once (like STL SET)
    X.insert(1); X.insert(18); X.insert(2); X.insert(2); X.insert(4); X.insert(8);
X.insert(16);

    // -------------- ITERATION --------------
    // Index-Wise : log_n
    cout << *X.find_by_order(0) << endl;
    cout << *X.find_by_order(2) << endl;
    cout << *X.find_by_order(6) << endl;    // Not Present, Will return 0

    if(X.end() == X.find_by_order(6)) cout << "End Reached" << endl;
    X.erase(X.find_by_order(2));        // Deleting element by iterator
    cout << "Iterating \n";
    for(auto x : X) cout << x << endl;
    cout << endl;

    // -------------- Range Search ------------
    // Returns number of elements STRICTLY less than value
    cout << X.order_of_key(-5) << endl;
    cout << X.order_of_key(1) << endl;
    cout << X.order_of_key(3) << endl;
    cout << X.order_of_key(4) << endl;
    cout << X.order_of_key(400) << endl;
    X.clear();          // Clearing
}

// Persistant/Dynamic Segment Tree
// Pointer Version

struct node {
    ll val;
    node *lft, *rht;
    node(node *L = NULL, node *R = NULL, ll v = 0) {
        lft = L;
        rht = R;
        val = v;
}};
node *persis[101000], *null = new node();
// null->lft = null->rht = null;
node *nCopy(node *x) {
    node *tmp = new node();
    if(x) {
        tmp->val = x->val;
        tmp->lft = x->lft;
        tmp->rht = x->rht;
    }
    return tmp;
}
void init(node *pos, ll l, ll r) {
    if(l == r) {
        pos->val = val[l];
        pos->lft = pos->rht = null;
        return;
    }
    ll mid = (l+r)>>1;
    pos->lft = new node();
    pos->rht = new node();
    init(pos->lft, l, mid);
    init(pos->rht, mid+1, r);
    pos->val = pos->lft->val + pos->rht->val;
}
// Single Position update
void update(node *pos, ll l, ll r, ll idx, ll val) {
    if(l == r) {
        pos->val += val;
        pos->lft = pos->rht = null;
        return;
    }
    ll mid = (l+r)>>1;
    if(idx <= mid) {
        pos->lft = nCopy(pos->lft);
        if(!pos->rht)
            pos->rht = null;
        update(pos->lft, l, mid, idx, val);
    }
    else {
        pos->rht = nCopy(pos->rht);
        if(!pos->lft)
            pos->lft = null;
        update(pos->rht, mid+1, r, idx, val);
    }
    pos->val = 0;
    if(pos->lft) pos->val += pos->lft->val;
    if(pos->rht) pos->val += pos->rht->val;
}
// Range [L, R] update
void update(node *pos, ll l, ll r, ll L, ll R, ll val) {
    if(r < L || R < l) return;
    if(L <= l && r <= R) {
        pos->prop += val;
        pos->val += (r-l+1)*val;
        return;
    }
    ll mid = (l+r)>>1;
    pos->lft = nCopy(pos->lft);              // Can be reduced
    pos->rht = nCopy(pos->rht);
    update(pos->lft, l, mid, L, R, val);
    update(pos->rht, mid+1, r, L, R, val);
    pos->val = pos->lft->val + pos->rht->val + (r-l+1)*pos->prop;
}
ll query(node *pos, ll l, ll r, ll L, ll R) {           // Range [L, R] Sum Query
    if(r < L || R < l || pos == NULL) return 0;
    if(L <= l && r <= R) return pos->val;
    ll mid = (l+r)/2LL;
    ll x = query(pos->lft, l, mid, L, R);
    ll y = query(pos->rht, mid+1, r, L, R);
    return x+y;
}
// Ignore LMax persistant tree positions query for finding k'th value
int query(node *RMax, node *LMax, int l, int r, int k) {                         // (LMax :
past, RMax : updated)
    if(l == r) return l;
    // NO NEED THIS SECTOR STILL AC --
    RMax->lft = nCopy(RMax->lft);
    LMax->lft = nCopy(LMax->lft);
    RMax->rht = nCopy(RMax->rht);
    LMax->rht = nCopy(LMax->rht);
    //--------------------------------
    // for each range [l, r] we will ignore every [1, l-1] range numbers
    int Count = RMax->lft->val - LMax->lft->val;
    int mid = (l+r)>>1;
    // if there exists more than or equal to k values in left range, then we'll find kth
number in left segment
    if(Count >= k)  return query(RMax->lft, LMax->lft, l, mid, k);
    else            return query(RMax->rht, LMax->rht, mid+1, r, k-Count);
}

// --------------- Propagation -------------------------
bool flipProp(bool par, bool child) {
    if(par == child) return 0;
    return 1;
}
void propagate(node *pos, ll l, ll r) {         // Propagation Func
    if(l == r) return;
    pos->lft = nCopy(pos->lft);                 // No need to copy in update/query function
    pos->rht = nCopy(pos->rht);
    if(!pos->flip) return;
    ll mid = (l+r)>>1;
    pos->lft->flip = flipProp(pos->flip, pos->lft->flip);
    pos->rht->flip = flipProp(pos->flip, pos->rht->flip);
    pos->lft->val = (mid-l+1)-pos->lft->val;
    pos->rht->val = (r-mid)-pos->rht->val;
    pos->flip = 0;
}
// Flip in range [L, R]
void updateFlip(node *pos, ll l, ll r, ll L, ll R) {
    if(r < L || R < l) return;
    propagate(pos, l, r);
    if(L <= l && r <= R) {
        pos->flip = 1;
        pos->val = (r-l+1) - pos->val;
        return;
    }
    ll mid = (l+r)>>1;
    updateFlip(pos->lft, l, mid, L, R);
    updateFlip(pos->rht, mid+1, r, L, R);
    pos->val = 0;
    if(pos->rht) pos->val += pos->rht->val;
    if(pos->lft) pos->val += pos->lft->val;
}
// Erasing A segment tree, pos = root, must run for each root
void ClearTree(node *pos) {
    if(pos == NULL) {
        delete pos;
        return;
    }
    ClearTree(pos->lft);
    ClearTree(pos->rht);
    delete pos;
}

int main() {
    // MUST BE INITIALIZED
    null->lft = null->rht = null;
    //
    for(int i = 1; i <= 10; ++i) {
        persis[i] = nCopy(persis[i-1]);
        update(persis[i], 1, n, idx, val);
    }
    return 0;
}

// Limit --------- No. of Primes
// 100             25
// 1000            168
// 10,000          1229
// 100,000         9592
// 1,000,000       78498
// 10,000,000      664579

bitset<10000000>isPrime;
vector<long long>primes;

void sieve(unsigned long long N) {
    isPrime.set();
    isPrime[0] = isPrime[1] = 0;
    unsigned long long lim = sqrt(N) + 5;
    for(unsigned long long i = 2; i <= lim; i++) {      // change lim to N, if all primes in range N is needed
        if(isPrime[i])
            for(unsigned long long j = i*i; j <= N; j+= i)
                isPrime[j] = 0;
}}

```

```
void sieveGen(unsigned long long N) {
    isPrime.set();
    isPrime[0] = isPrime[1] = 0;
    for(unsigned long long i = 2; i <= N; i++) {        //Note, N isn't square rooted!
        if(isPrime[i]) {
            for(unsigned long long j = i*i; j <= N; j+= i)
                isPrime[j] = 0;
            primes.push_back(i);
}}}

vector<pair<ull, ull> > primeFactor(ull n) {
    vector<pair<ull, ull> >factor;
    for(long long i = 0; i < (int)primes.size() && primes[i] <= n; i++) {
        bool first = 1;
        while(n%primes[i] == 0) {
            if(first) {
                factor.push_back({primes[i], 0});
                first = 0;
            }
            factor.back().second++;
            n/=primes[i];
    }}
    return factor;
}

int pd[MAX];                    // Contains minimum prime factor/divisor, for primes
pd[x] = x
vector<int>primes;              // Contains prime numbers
void SieveLinear(int N) {
    for(int i = 2; i <= N; ++i) {
        if(pd[i] == 0) pd[i] = i, primes.push_back(i);                  // if pd[i] == 0,
then i is prime
        for(int j=0; j < (int)primes.size() && primes[j] <= pd[i] && i*primes[j] <= N; ++j)
            pd[i*primes[j]] = primes[j];
}}

int pd[MAX];                    // Contains minimum prime factor/divisor, for primes
pd[x] = x
vector<int>primes;              // Contains prime numbers
vector<int>PF[MAX];
void SieveLinearRangePF(int N, ll low, ll hi) {             // also returns unique prime
factors in range [low, hi]
    for(int i = 2; i <= N; ++i) {
        if(pd[i] == 0) {
            pd[i] = i, primes.push_back(i);                     // if pd[i] == 0, then i is
prime
            for(ll x = (low-1)-(low-1)%i+i; x <= hi; x += i)        // inserting all prime
factors [prime will be inserted only once]
                PF[x-low].push_back(i);                             // just to be sure, used
low-1, instead of low
        }
        for(int j=0; j < (int)primes.size() && primes[j] <= pd[i] && i*primes[j] <= N; ++j)
            pd[i*primes[j]] = primes[j];
}}

vector<ll> Divisors(ll n) {                     // Returns the divisors
    ll lim = sqrt(n);
    vector<ll>divisor;
    for(ll i = 2; i <= lim; i++) {          // deal with 1 and n manually
        if(n % i == 0) {
            ll tmp = n/i;
            divisor.push_back(tmp);
            if(i != tmp)
                divisor.push_back(i);
    }}
    return divisor;
}

vector<pair<long long, long long> > factorialFactorization(long long n) {   // prime factorization of factorials (n!)
    vector<pair<long long, long long> >V;
    for(long long i = 0; i < (int)primes.size() && primes[i] <= n; i++) {
        long long tmp = n, power = 0;
        while(tmp/primes[i]) {
            power += tmp/primes[i];
            tmp /= primes[i];
        }
        if(power != 0) V.push_back(make_pair(primes[i], power));
    }
    return V;
}

long long numPF(long long n) {          //returns number of prime factors
    long long num = 0;
    for(long long i = 0; primes[i] * primes[i] <= n; i++) {
        while(n % primes[i] == 0) {
            n /= primes[i];
            num++;
    }}
    if(n > 1) num++;        //there might left a prime number which is bigger than primes[i]
    return num;
}

long long numDIFPF(long long n) {       // returns number of different prime factors
    long long diff_num = 0;
    for(long long i = 0; primes[i] * primes[i] <= n; i++) {
        bool ok = 0;
        while(n % primes[i] == 0) {
            n /= primes[i];
            ok = 1;
        }
        if(ok) diff_num++;
    }
    if(n > 1) diff_num++;
    return diff_num;
}

unsigned long long sumPF(long long n) {     // returns sum of prime factors
    unsigned long long sum = 0;
    for(long long i = 0; primes[i] * primes[i] <= n; i++)
        while(n % primes[i] == 0) {
            n /= primes[i];
            sum+=primes[i];
        }
    if(n > 1) sum+= n;
    return sum;
```

```
}

int NumberOfDivisors(long long n) {       // if n = p1^a1 * p2^a2,... then NOD = (a1+1)*(a2+1)*...
    if(n <= MAX and isPrime[n]) return 2;
    int NOD = 1;
    for(int i = 0, a = 0; i < (int)primes.size() and primes[i] <= n; ++i, a = 0) {
        while(n % primes[i] == 0)
            ++a, n /= primes[i];
        NOD *= (a+1);
    }
    if(n != 1) NOD *= 2;
    return NOD;
}

//------Fast Factorization using Sieve-Like algorithm-------
bitset<MAX>isPrime;
int divisor[MAX];

void sieve(long long lim) {             // Prime numbers for the limit should be sieved, otherwise WA
    isPrime.set();
    isPrime[0] = isPrime[1] = 0;
    for(ll i = 0; i <= lim; ++i) {
        if(isPrime[i]) {
            for(long long j = i; j <= lim; j += i) {
                isPrime[j] = 0;
                divisor[j] = i;
}}}}

vector<int> factorize(long long x) {    // This function only iterates over the prime numbers
    int pastDiv = 0;                    // 0 : no divisor is present
    vector<int>factor;
    while(x > 1) {
        if(divisor[x] != 0) {
            factor.push_back(divisor[x]);
            x /= divisor[x];            // now x would be reduced by factor of divisor[x]
    }}
    return factor;
}
//------------------------------------------------------

// Prime Probability
// Algorithm : Miller-Rabin primality test       Complexity : k * (log n)^3
// This function is called for all k trials. It returns false if n is composite and returns false if n is probably prime.
// d is an odd number such that  d*(2^r) = n-1 for some r >= 1

bool miillerTest(int d, int n) {
    int a = 2 + rand() % (n - 4);       // Pick a random number in [2..n-2].
    int x = Pow(a, d, n);               // Compute a^d % n
    if (x == 1  || x == n-1)
       return 1;
    while (d != n-1) {                      // Keep squaring x while one of the following doesn't happen
        x = (x * x) % n;                    // (i)   d does not reach n-1
        d *= 2;                             // (ii)  (x^2) % n is not 1
        if (x == 1)      return 0;          // (iii) (x^2) % n is not n-1
        if (x == n-1)    return 1;
    }
    return 0;               // Return composite
}

bool isPrime(int n, int k = 10) {               // Higher value of k gives more accuracy (Use k >= 9)
    if(n <= 1 || n == 4)  return 0;             // Corner cases
    if(n <= 3) return 1;
    int d = n - 1;                              // Find r such that n = 2^d * r + 1 for some r >= 1
    while(d % 2 == 0)  d /= 2;
    for(int i = 0; i < k; i++)                  // Iterate given nber of 'k' times
        if(miillerTest(d, n) == 0)
            return 0;
    return 1;
}

// Binary Search
// Complexity : O(n Log n)

ll UpperBound(ll lo, ll hi, ll key) {               // Returns lowest position where v[i] > key
    ll mid, ans = -1;                               // 10 10 10 20 20 20 30 30
    while(lo <= hi) {                               //                   ^
        mid = (lo + hi)>>1;
        if(key >= v[mid])   ans = mid, lo = mid + 1;
        else                hi = mid - 1;
    }
    return ans+1;                               // Tweaking this line will return the last position of key
}

ll LowerBound(ll lo, ll hi, ll key) {           // Returns lowest position where v[i] == key (if value is present more than once)
    ll mid, ans = -1;                           // 10 10 10 20 20 20 30 30
    while(lo <= hi) {                           //           ^
        mid = (lo+hi)>>1;
        if(key <= v[mid])   ans = mid, hi = mid - 1;
        else                lo = mid + 1;
    }
    return ans;
}

// lo : lower value, hi : upper value, est : estimated output of the required result, delta : number of iteration in search
double bisection(double lo, double hi, double est, int delta) {
    double mid, ans = -1;
    for(int i = 0; i < delta; ++i) {
        mid = (lo+hi)/2.0;
        if(Equal(TestFunction(mid), est))           ans = mid, lo = mid;
        else if(Greater(TestFunction(mid), est))    hi = mid;
        else                                        lo = mid;
    }
    return ans;
}

```

```
// Full Functional Ternary Search
/* EMAXX ::
If f(x) takes integer parameter, the interval [l r] becomes discrete.
Since we did not impose any restrictions on the choice of points m1 and m2, the correctness
of the algorithm is not affected.
m1 and m2 can still be chosen to divide [l r] into 3 approximately equal parts.

The difference occurs in the stopping criterion of the algorithm.
Ternary search will have to stop when (r−l) < 3, because in that case we can no longer
select m1 and m2 to
be different from each other as well as from ll and rr, and this can cause infinite
iterating.
Once (r−l) < 3, the remaining pool of candidate points (l,l+1,…,r) needs to be checked
to find the point which produces the maximum value f(x).
*/

ll ternarySearch(ll low, ll high) {
    ll ret = -INF;
    while((high - low) > 2) {
        ll mid1 = low + (high - low) / 3;
        ll mid2 = high - (high - low) / 3;
        ll cost1 = f(mid1);
        ll cost2 = f(mid2);
        if(cost1 < cost2) {
            low = mid1;
            ret = max(cost2, ret);
        }
        else {
            high = mid2;
            ret = max(cost1, ret);
    }}
    for(int i = low; i <= high; ++i)
        ret = max(ret, f(i));
    return ret;
}

// Segment Tree

// Only Supports Range Value SET (NOT UPDATE) and Point Query
struct SegTreeSetVal {
    vector<int>tree;
    vector<bool>prop;

    void Resize(int n) {
        tree.resize(n*5);
        prop.resize(n*5);
    }

    void propagate(int pos, int l, int r) {
        if(!prop[pos] || l == r) return;
        tree[pos<<1|1] = tree[pos<<1] = tree[pos];
        prop[pos<<1|1] = prop[pos<<1] = 1;
        prop[pos] = 0;
    }

    void SetVal(int pos, int l, int r, int L, int R, int val) {     // Set value val in
range [L, R]
        if(r < L || R < l) return;
        propagate(pos, l, r);
        if(L <= l && r <= R) {
            tree[pos] = val;
            prop[pos] = 1;
            return;
        }
        int mid = (l+r)>>1;
        SetVal(pos<<1, l, mid, L, R, val);
        SetVal(pos<<1|1, mid+1, r, L, R, val);
    }

    int query(int pos, int l, int r, int idx) {             // Can be modified to range
query
        if(l == r) return tree[pos];
        propagate(pos, l, r);
        int mid = (l+r)>>1;
        if(idx <= mid)  return query(pos<<1, l, mid, idx);
        else            return query(pos<<1|1, mid+1, r, idx);
}};


// Segment Tree Range Sum : Lazy with Propagation (MOD used)
struct SegTreeRSQ {
    vector<ll>sum, prop;

    void Resize(int n) {
        sum.resize(5*n);
        prop.resize(5*n);
    }

    void init(int pos, int l, int r, ll val[]) {
        sum[pos] = prop[pos] = 0;
        if(l == r) {
            sum[pos] = val[l]%MOD;
            return;
        }
        int mid = (l+r)>>1;
        init(pos<<1, l, mid, val);
        init(pos<<1|1, mid+1, r, val);
        sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
    }

    void propagate(int pos, int l, int r) {
        if(prop[pos] == 0 || l == r) return;
        int mid = (l+r)>>1;

        sum[pos<<1] = (sum[pos<<1] + prop[pos]*(mid-l+1))%MOD;
        sum[pos<<1|1] = (sum[pos<<1|1] + prop[pos]*(r-mid))%MOD;
        prop[pos<<1] = (prop[pos<<1] + prop[pos])%MOD;
        prop[pos<<1|1] = (prop[pos<<1|1] + prop[pos])%MOD;
        prop[pos] = 0;
    }

    void update(int pos, int l, int r, int L, int R, ll val) {
        if(r < L || R < l) return;
        propagate(pos, l, r);
        if(L <= l && r <= R) {
            sum[pos] = (sum[pos] + val*(r-l+1))%MOD;
            prop[pos] = (prop[pos] + val)%MOD;
            return;
        }

        int mid = (l+r)>>1;
        update(pos<<1, l, mid, L, R, val);
        update(pos<<1|1, mid+1, r, L, R, val);
        sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
    }

    ll query(int pos, int l, int r, int L, int R) {
        if(r < L || R < l) return 0;
        propagate(pos, l, r);
        if(L <= l && r <= R) return sum[pos];
        int mid = (l+r)>>1;
        return (query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R))%MOD;
}};


// Dynamic Range Segment Tree (values can be deleted from right)
// Resize the Segment Tree with the maximum length of value
// Segment Tree Range Sum, Range Update, And Single point Value Change (If the last value was deleted)
// http://codeforces.com/contest/283/problem/A

// Range Sum with Carry Value
struct SegSum {
    int tree[MAX*4], carry[MAX*4];

    void init() {
        memset(tree, 0, sizeof tree);
        memset(carry, 0, sizeof carry);
    }

    void Update(int pos, int l, int r, int x, int y, ll val) {              // Update value at range/point
        if(y < l || x > r) return;
        if(x <= l && r <= y) {
            tree[pos] += (r-l+1)*val;
            carry[pos] += val;
            return;
        }
        int mid = (l+r)>>1;
        Update(pos<<1, l, mid, x, y, val);
        Update(pos<<1|1, mid+1, r, x, y, val);
        tree[pos] = tree[pos<<1] + tree[pos<<1|1] + (r-l+1)*carry[pos];
    }

    ll Read(int pos, int l, int r, int x, int y, ll Carry = 0) {            // Read value at range/point
        if(y < l || x > r)      return 0;
        if(x <= l && r <= y)    return tree[pos] + Carry * (r-l+1);
        ll mid = (l+r)>>1;
        ll lft = Read(pos<<1, l, mid, x, y, Carry + carry[pos]);
        ll rht = Read(pos<<1|1, mid+1, r, x, y, Carry + carry[pos]);
        return lft + rht;
    }

    // Sets value at idx
    void Set(int pos, int l, int r, int idx, ll val, ll Carry = 0) {
        if(l == r) {
            tree[pos] = val + (-1*Carry);   // Extra carry values are eleminated in such way
            carry[pos] = 0;                 // that the subtraction is always the new value
            return;
        }
        int mid = (l+r)>>1;
        if(idx <= mid)  Set(pos<<1, l, mid, idx, val, Carry + carry[pos]);
        else            Set(pos<<1|1, mid+1, r, idx, val, Carry + carry[pos]);
        tree[pos] = tree[pos<<1] + tree[pos<<1|1] + (r-l+1) * carry[pos];
}};

// SegTree with Lazy Propagation (Flip Count in Range)
// Prop :
// 0 : No prop operation
// 1 : Prop operation should be done

struct SegProp {
    struct Node { int val, prop; };

    vector<Node>tree;
    void init(int L, int R, int pos, ll val[]) {
        if(L == R) {
            tree[pos].val = 0;
            tree[pos].prop = 0;
            return;
        }

        int mid = (L+R)>>1;
        init(L, mid, pos<<1, val);
        init(mid+1, R, pos<<1|1, val);
        tree[pos].val = tree[pos].prop = 0;
    }

    int flipProp(int parentVal, int childVal) {
        if(parentVal == childVal) return 0;
        return parentVal;
    }

    void propagate(int L, int R, int pos) {
        if(tree[pos].prop == 0 || L == R)       // If no propagation tag
            return;                             // or leaf node, then no need to change
        int mid = (L+R)>>1;
        tree[pos<<1].val = (mid-L+1) - tree[pos<<1].val;                      // Set left & right child value
        tree[pos<<1|1].val = (R-mid) - tree[pos<<1|1].val;
        tree[pos<<1].prop = flipProp(tree[pos].prop, tree[pos<<1].prop);    // Flip child prop according to problem
        tree[pos<<1|1].prop = flipProp(tree[pos].prop, tree[pos<<1|1].prop);
        tree[pos].prop = 0;                                                 // Clear parent propagation tag
    }

    void update(int L, int R, int l, int r, int pos) {
        if(r < L || R < l) return;
        propagate(L, R, pos);
        if(l <= L && R <= r) {
            tree[pos].val = (R-L+1) - tree[pos].val;    // Value updated
            tree[pos].prop = 1;                         // Propagation tag set
            return;
        }
        int mid = (L+R)>>1;
        update(L, mid, l, r, pos<<1);
        update(mid+1, R, l, r, pos<<1|1);
        tree[pos].val = tree[pos<<1].val + tree[pos<<1|1].val;
    }

    int querySum(int L, int R, int l, int r, int pos) {
        if(r < L || R < l) return 0;
        propagate(L, R, pos);
        if(l <= L && R <= r) return tree[pos].val;
        int mid = (L+R)>>1;
        int lft = querySum(L, mid, l, r, pos<<1);
        int rht = querySum(mid+1, R, l, r, pos<<1|1);
        return lft+rht;
}};

// ----------------------- Segment Tree Range Maximum Sum ----------------------
struct SegTreeRMS {
    struct node {
        ll sum, prefix, suffix, ans;

        node(ll val = 0) {
            sum = prefix = suffix = ans = val;
        }

        void merge(node left, node right) {
            sum = left.sum + right.sum;
            prefix = max(left.prefix, left.sum+right.prefix);
            suffix = max(right.suffix, right.sum+left.suffix);
            ans = max(left.ans, max(right.ans, left.suffix+right.prefix));
    }};

    vector<node>tree;
    void init(int pos, int l, int r, ll val[]) {
        if(l == r) {
            tree[pos] = node(val[l]);
            return;
        }
        int mid = (l+r)/2;
        init(pos*2, l, mid, val);
        init(pos*2+1, mid+1, r, val);
        tree[pos] = node(-INF);
        tree[pos].merge(tree[pos*2], tree[pos*2+1]);
    }

    void update(int pos, int l, int r, int x, int val) {
        if(x < l || r < x) return;
        if(l == r && l == x) {
            tree[pos] = node(val);
            return;
        }
        int mid = (l+r)/2;
        update(pos*2, l, mid, x, val);
        update(pos*2+1, mid+1, r, x, val);
        tree[pos] = node(-INF);
        tree[pos].merge(tree[pos*2], tree[pos*2+1]);
    }

    node query(int pos, int l, int r, int x, int y) {
        if(r < x || y < l)      return node(-INF);
        if(x <= l && r <= y)    return tree[pos];
        int mid = (l+r)/2;
        node lft = query(pos*2, l, mid, x, y);
        node rht = query(pos*2+1, mid+1, r, x, y);
        node parent = node(-INF);
        parent.merge(lft, rht);
        return parent;
}};

// Segment Tree Insert/Remove value, Find I'th Value
struct SegTreeInsertRemove {                    // Finds/Deletes I'th value from
array/SegTree
    int tree[MAX*4];
    void init(int pos, int L, int R) {
        if(L == R) {
            tree[pos] = 1;
            return;
        }
        int mid = (L+R)>>1;
        init(pos<<1, L, mid);
        init(pos<<1|1, mid+1, R);
        tree[pos] = tree[pos<<1]+tree[pos<<1|1];
    }
    int SearchVal(int pos, int L, int R, int I, bool removeVal = 0) {         // Find I'th
value in Segment Tree, removes it if removeVal = 1
        if(L == R) {
            tree[pos] = (removeVal ? 0:1);
            return L;
        }
        int mid = (L+R)>>1;
        if(I <= tree[pos<<1]) {
            int idx = SearchVal(pos<<1, L, mid, I, removeVal);
            if(removeVal) tree[pos] = tree[pos<<1] + tree[pos<<1|1];
            return idx;
        }
        else {
            int idx = SearchVal(pos<<1|1, mid+1, R, I-tree[pos<<1], removeVal);
            if(removeVal) tree[pos] = tree[pos<<1] + tree[pos<<1|1];
            return idx;
}}};

// Segment Tree Range Bit flip, set, reset and Query
// propataion tags:
// 0 - no change, 1 - all set to one, 2 - all set to zero, 3 - all need to be flipped
struct RangeBitQuery {
    vector<pair<int, int> >tree;
    RangeBitQuery() { tree.resize(MAX*4); }

    void init(int pos, int L, int R, string &s) {
        tree[pos].second = 0;
        if(L == R) {
            tree[pos].first = (s[L] == '1');
            return;
        }
        int mid = (L+R)>>1;
        init(pos<<1, L, mid, s);
        init(pos<<1|1, mid+1, R, s);
        tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
    }

    int Convert(int tag) {          // This function generates output tag of child node if the parent node is set to 3 (fipped)
        if(tag == 1) return 2;
        if(tag == 2) return 1;
        if(tag == 3) return 0;
        return 3;
    }

    // On every layer of update or query, this Propagation func should be called to pre-process previous left off operations
    void Propagate(int L, int R, int parent) {      // Propagate parent node to child nodes (left and right)
        if(tree[parent].second == 0) return;        // and sets parent node's propagation tag to 0
        int mid = (L+R)>>1;
        int lft = parent<<1, rht = parent<<1|1;
        if(tree[parent].second == 1) {
            tree[lft].first = mid-L+1;
            tree[rht].first = R-mid;
        }
        else if(tree[parent].second == 2)
            tree[lft].first = tree[rht].first = 0;
        else if(tree[parent].second == 3) {
            tree[lft].first = (mid-L+1) - tree[lft].first;
            tree[rht].first = (R-mid) - tree[rht].first;
        }
        if(L != R) {                                // If the child nodes also contain propagate tag (and the childs are not leaf node)
            if(tree[parent].second == 1 || tree[parent].second == 2)
                tree[lft].second = tree[rht].second = tree[parent].second;
            else {
                tree[lft].second = Convert(tree[lft].second);
                tree[rht].second = Convert(tree[rht].second);
        }}
        tree[parent].second = 0;                                          // Parent node's prop tag set to zero
        if(L!=R) tree[parent].first = tree[lft].first + tree[rht].first;    // If this is not the leaf node, calculate child node's sum
    }

    void updateOn(int pos, int L, int R, int l, int r) {                // Turn on bits in range [l, r]
        if(r < L || R < l || L > R) return;
        Propagate(L, R, pos);
        if(l <= L && R <= r) {
            tree[pos].first = (R-L+1);
            tree[pos].second = 1;
            return;
        }
        int mid = (L+R)>>1;
        updateOn(pos<<1, L, mid, l, r);
        updateOn(pos<<1|1, mid+1, R, l, r);
        tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
    }

    void updateOff(int pos, int L, int R, int l, int r) {               // Turn off bits in range [l, r]
        if(r < L || R < l || L > R) return;
        Propagate(L, R, pos);
        if(l <= L && R <= r) {
            tree[pos].first = 0;
            tree[pos].second = 2;
            return;
        }
        int mid = (L+R)>>1;
        updateOff(pos<<1, L, mid, l, r);
        updateOff(pos<<1|1, mid+1, R, l, r);
        tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
    }

    void updateFlip(int pos, int L, int R, int l, int r) {              // Flip bits in range [l, r]
        if(r < L || R < l || L > R) return;
        Propagate(L, R, pos);
        if(l <= L && R <= r) {
            tree[pos].first = abs(R-L+1 - tree[pos].first);
            tree[pos].second = 3;
            return;
        }
        int mid = (L+R)>>1;
        updateFlip(pos<<1, L, mid, l, r);
        updateFlip(pos<<1|1, mid+1, R, l, r);
        tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
    }

    int querySum(int pos, int L, int R, int l, int r) {                 // Returns number of set bit in range [l, r]
        if(r < L || R < l || L > R) return 0;
        Propagate(L, R, pos);
        if(l <= L && R <= r) return tree[pos].first;
        int mid = (L+R)>>1;
        return querySum(pos<<1, L, mid, l, r) + querySum(pos<<1|1, mid+1, R, l, r);
}};

// Merge Sort Tree
struct MergeSortTree {
    vector<int>tree[MAX*4];

    void init(int pos, int l, int r, ll val[]) {
        tree[pos].clear();                                // Clears past values
        if(l == r) {
            tree[pos].push_back(val[l]);
            return;
        }

        int mid = (l+r)>>1;
        init(pos<<1, l, mid, val);
        init(pos<<1|1, mid+1, r, val);
        merge(tree[pos<<1].begin(), tree[pos<<1].end(), tree[pos<<1|1].begin(),
tree[pos<<1|1].end(), back_inserter(tree[pos]));
    }

    int query(int pos, int l, int r, int L, int R, int k) {
        if(r < L || R < l) return 0;
        if(L <= l && r <= R)
            return (int)tree[pos].size() - (upper_bound(tree[pos].begin(), tree[pos].end(),
k) - tree[pos].begin());        // MODIFY
        int mid = (l+r)>>1;
        return query(pos<<1, l, mid, L, R, k) + query(pos<<1|1, mid+1, r, L, R, k);
}};


// Segment Tree Sequence (Lazy Propagation):: Contains sequnce A + 2A + 3A + ..... nA
struct SegTreeSeq {
    vector<ll>sum, prop;

    void Resize(int n) {
        sum.resize(n*5);
        prop.resize(n*5);
    }

    ll intervalSum(ll l, ll r, ll val) {
        ll interval = (r*(r+1))/2LL - (l*(l-1))/2LL;
        return (interval*val+MOD)%MOD;
    }

    void propagate(int pos, int l, int r) {
        if(prop[pos] == 0 || l == r) return;
        int mid = (l+r)>>1;

        sum[pos<<1] = (sum[pos<<1] + intervalSum(l, mid, prop[pos]))%MOD;
        sum[pos<<1|1] = (sum[pos<<1|1] + intervalSum(mid+1, r, prop[pos]))%MOD;
        prop[pos<<1] = (prop[pos<<1] + prop[pos])%MOD;
        prop[pos<<1|1] = (prop[pos<<1|1] + prop[pos])%MOD;
        prop[pos] = 0;
    }

    void init(int pos, int l, int r, ll val[]) {
        sum[pos] = prop[pos] = 0;
        if(l == r) {
            sum[pos] = (val[l]*l)%MOD;
            return;
        }
        int mid = (l+r)>>1;
        init(pos<<1, l, mid, val);
        init(pos<<1|1, mid+1, r, val);
        sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
    }

    void update(int pos, int l, int r, int L, int R, ll val) {      // Range Update
        if(r < L || R < l) return;
        propagate(pos, l, r);
        if(L <= l && r <= R) {
            sum[pos] = (intervalSum(l, r, val) + sum[pos])%MOD;
            prop[pos] = (val + prop[pos])%MOD;
            return;
        }
        int mid = (l+r)>>1;
        update(pos<<1, l, mid, L, R, val);
        update(pos<<1|1, mid+1, r, L, R, val);
        sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
    }

    ll query(int pos, int l, int r, int L, int R) {     // Range Query
        if(r < L || R < l || L > R) return 0;
        propagate(pos, l, r);
        if(L <= l && r <= R) return sum[pos];
        int mid = (l+r)>>1;
        return (query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R))%MOD;
}};

// Segment Tree Bracket Sequencing, Modify position bracket and check if it is valid
struct BracketTree {
    struct node{
        int BrcStart, BrcEnd;               // number of start bracket, number of end bracket
        bool isOk = 0;                      // is the sequence valid

        node(int a = 0, int b = 0) {
            BrcStart = a;
            BrcEnd = b;
            isOk = (BrcStart == 0 && BrcEnd == 0);
        }
        node(char c) {
            if(c == '(')    BrcStart = 1, BrcEnd = 0;
            else            BrcStart = 0, BrcEnd = 1;
        }
        void mergeNode(node lft, node rht) {
            if(lft.isOk && rht.isOk)
                BrcStart = 0, BrcEnd = 0, isOk = 1;
            else {
                int match = min(lft.BrcStart, rht.BrcEnd);
                BrcStart = lft.BrcStart - match + rht.BrcStart;
                BrcEnd = lft.BrcEnd + rht.BrcEnd - match;
                (BrcStart == 0 && BrcEnd == 0) ? isOk = 1: isOk = 0;
    }}};

    node tree[MAX*4];
    void init(int pos, int L, int R, char s[]) {
        if(L == R) {
            tree[pos] = node(s[L]);
            return;
        }
        int mid = (L+R)>>1;
        init(pos<<1, L, mid, s);
        init(pos<<1|1, mid+1, R, s);
        tree[pos].mergeNode(tree[pos<<1], tree[pos<<1|1]);
    }
    void update(int pos, int L, int R, int idx, char val) {         // idx : index of the changed value
        if(idx < L || R < idx) return;                              // val : changed bracket sequence in char ( or )
        if(L == R && L == idx) {
            tree[pos] = node(val);
            return;
        }
        int mid = (L+R)>>1;
        update(pos<<1, L, mid, idx, val);
        update(pos<<1|1, mid+1, R, idx, val);
        tree[pos].mergeNode(tree[pos<<1], tree[pos<<1|1]);
    }
    bool isValid() {                        // Returns True if sequence is valid
        return tree[1].isOk;
}};

// Outputs Largest Balanced Bracket Sequence in range [L, R]
struct MaxBracketSeq {
    struct node {
        ll lftBracket, rhtBracket, Max;
        node(ll lft=0, ll rht=0, ll Max=0) {
            this->lftBracket = lft;
            this->rhtBracket = rht;
            this->Max = Max;
    }};

    node tree[MAX*4];
    node Merge(const node &lft, const node &rht) {
        ll common = min(lft.lftBracket, rht.rhtBracket);
        ll lftBracket = lft.lftBracket + rht.lftBracket - common;
        ll rhtBracket = lft.rhtBracket + rht.rhtBracket - common;
        return node(lftBracket, rhtBracket, lft.Max+rht.Max+common);
    }

    void init(ll pos, ll l, ll r, char s[]) {
        if(l == r) {
            if(s[l-1] == '(')   tree[pos] = node(1, 0, 0);
            else                tree[pos] = node(0, 1, 0);
            return;
        }
        ll mid = (l+r)>>1;
        init(pos<<1, l, mid, s);
        init(pos<<1|1, mid+1, r, s);
        tree[pos] = Merge(tree[pos<<1], tree[pos<<1|1]);
    }

    node query(ll pos, ll l, ll r, ll L, ll R) {
        if(r < L || R < l)      return node();
        if(L <= l && r <= R)    return tree[pos];
        ll mid = (l+r)>>1;
        node lft = query(pos<<1, l, mid, L, R);
        node rht = query(pos<<1|1, mid+1, r, L, R);
        return Merge(lft, rht);
    }

    int MaxSequence(int SEQ_SIZE, int l, int r) {
        return 2*query(1, 1, SEQ_SIZE, l, r).Max;
}};

// Path Compression Basics
// in segment tree comparison of index must be checked like (where l, r is the query range):
// outside of range [l, r]  :  r < point[L] || point[R] < l
// inside of range  [l, r]  :  l <= point[L] && point[R] <= r
// The Queries {l, r} will be in a queue, and processed after CompressPath and
initialization is done

void CompressPath(vector<int> &point) {                                     // point contains
all left and right boundary and query boundaries
    point.push_back(0);                                                     // push_back a
minimum value which is lower than input values
    sort(point.begin(), point.end());                                       // so that the input
values start from index 1
    point.erase(unique(point.begin()+1, point.end()), point.end());     // Only unique
points taken, this will be the compressed points
}

// Finding Number of Uniques in Range + OFFLINE processing

struct FindUnique {
    int tree[4*MAX], prop[4*MAX], v[MAX], IDX[MAX];
    map<int, vector<int> >Map;
    map<pair<int, int>, int>Ans;
    vector<pair<int, int> > Query;

    void init() {
        memset(IDX, -1, sizeof IDX);
        memset(tree, 0, sizeof tree);
        Ans.clear(), Map.clear(), Query.clear();
    }
    void update(int pos, int L, int R, int idx, int val) {
        if(idx < L || R < idx) return;
        if(L == R) {
            tree[pos]+= val;
            return;
        }
        int mid = (L+R)>>1;
        update(pos<<1, L, mid, idx, val);
        update(pos<<1|1, mid+1, R, idx, val);
        tree[pos] = tree[pos<<1] + tree[pos<<1|1];
    }
    int query(int pos, int L, int R, int l, int r) {
        if(r < L || R < l)      return 0;
        if(l <= L && R <= r)    return tree[pos];
        int mid = (L+R)>>1;
        int lft = query(pos<<1, L, mid, l, r);
        int rht = query(pos<<1|1, mid+1, R, l, r);
        return lft+rht;
    }
    void ArrayInput(int SZ) {
        for(int i = 1; i <= SZ; ++i) scanf("%d", &v[i]);
    }
    void QueryInput(int q) {
        int l, r;
        while(q--) {
            scanf("%d %d", &l, &r);
            Query.push_back(make_pair(l, r));
            Map[r].push_back(l);                            // Used for sorting
    }}
    void GenAns(int SZ) {
        map<int, vi> :: iterator it;
        int lPos = 0;
        for(it = Map.begin(); it != Map.end(); ++it) {      // For each query's right points
            while(lPos < it->first) {                       // Update from last left position to this queries right position
                lPos++;
                if(IDX[v[lPos]] == -1) {
                    IDX[v[lPos]] = lPos;
                    update(1, 1, SZ, lPos, 1);              // if new value found, increment 1 to the
                }
                else {
                    int pastIDX = IDX[v[lPos]];
                    IDX[v[lPos]] = lPos;
                    update(1, 1, SZ, pastIDX, -1);           // if value found previous, then remove 1 from previous index (add -1)
                    update(1, 1, SZ, lPos, 1);               // add 1 to the new position
            }}
            for(int i = 0; i < (int)(it->second).size(); ++i)       // Range sum query for all queries that ends on this point
                Ans[make_pair(it->second[i], it->first)] = query(1, 1, SZ, it->second[i], it->first);
    }}
    void PrintAns() {
        for(int i = 0; i < (int)Query.size(); ++i)                    // Output according to input query
            printf("%d\n", Ans[mp(Query[i].first, Query[i].second)]);
}};

struct STreeMultipleOf3 {
    int tree[4*MAX][3], prop[4*MAX];
    void init(int pos, int L, int R) {
        if(L == R) {
            tree[pos][0] = 1, tree[pos][1] = tree[pos][2] = 0;
            return;
        }
        int mid = (L+R)>>1;
        init(pos<<1, L, mid);
        init(pos<<1|1, mid+1, R);
        for(int i = 0; i < 3; ++i)
            tree[pos][i] = tree[pos<<1][i] + tree[pos<<1|1][i];
    }
    void shiftVal(int pos, int step) {
        step %= 3;
        if(step == 0) return;
        swap(tree[pos][2], tree[pos][1]);
        swap(tree[pos][1], tree[pos][0]);
        if(step == 2) {
            swap(tree[pos][2], tree[pos][1]);
            swap(tree[pos][1], tree[pos][0]);
    }}
    void propagate(int pos, int L, int R) {
        if(L == R || prop[pos] == 0) return;
        shiftVal(pos<<1, prop[pos]), shiftVal(pos<<1|1, prop[pos]);
        prop[pos<<1] += prop[pos], prop[pos<<1|1] += prop[pos];
        prop[pos] = 0;
    }
    void update(int pos, int L, int R, int l, int r) {              // update l to r by 1
        if(r < L || R < l) return;
        if(prop[pos] != 0) propagate(pos, L, R);
        if(l <= L && R <= r) {
            shiftVal(pos, 1);
            prop[pos] += 1;
            return;
        }
        int mid = (L+R)>>1;
        update(pos<<1, L, mid, l, r);
        update(pos<<1|1, mid+1, R, l, r);
        for(int i = 0; i < 3; ++i)
            tree[pos][i] = tree[pos<<1][i] + tree[pos<<1|1][i];
    }
    int query(int pos, int L, int R, int l, int r) {                // return number of
multiple of 3 in range l to r
        if(r < L || R < l) return 0;
        propagate(pos, L, R);
        if(l <= L && R <= r) return tree[pos][0];
        int mid = (L+R)>>1;
        int lft = query(pos<<1, L, mid, l, r);
        int rht = query(pos<<1|1, mid+1, R, l, r);
        return lft+rht;
}};

// CS Academy Candles :https://csacademy.com/contest/archive/task/candles/statement/
struct SortedST {                                       // Performs -1 from n nodes and
keeps nodes sorted (descending order)
    struct node { int val, prop; };
    node tree[5*MAX];
    void init(int pos, int l, int r, int val[]) {       // val[] must be sorted from hi to
low
        if(l == r) {
            tree[pos].val = val[l];
            tree[pos].prop = 0;
            return;
        }
        int mid = (l+r)>>1;
        init(pos<<1, l, mid, val), init(pos<<1|1, mid+1, r, val);
        tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
        tree[pos].prop = 0;
    }
    void propagate(int pos, int l, int r) {
        if(tree[pos].prop == 0 || l == r) {
            tree[pos].prop = 0;
            return;
        }
        tree[pos<<1|1].prop += tree[pos].prop;
        tree[pos<<1].prop += tree[pos].prop;
        tree[pos<<1].val += tree[pos].prop;
        tree[pos<<1|1].val += tree[pos].prop;
        tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
        tree[pos].prop = 0;
    }
    int findVal(int pos, int l, int r, int idx) {                // Finds value in index idx
        if(l == r) return tree[pos].val;
        propagate(pos, l, r);
        int mid = (l+r)>>1;
        if(idx <= mid)  return findVal(pos<<1, l, mid, idx);
        else            return findVal(pos<<1|1, mid+1, r, idx);
    }
    void update(int pos, int l, int r, int L, int R, int val) {
        if(r < L || R < l) return;
        propagate(pos, l, r);
        if(L <= l && r <= R) {
            tree[pos].val += val;
            tree[pos].prop += val;
            return;
        }
        int mid = (l+r)>>1;
        update(pos<<1, l, mid, L, R, val);
        update(pos<<1|1, mid+1, r, L, R, val);
        tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
    }
    int rightMost(int pos, int l, int r, int val) {             // Finds rightmost value in tree that contains val
        if(l == r) return l;
        if(tree[pos].val < val) return 0;
        propagate(pos, l, r);
        int mid = (l+r)>>1;
        if(tree[pos<<1|1].val >= val) return rightMost(pos<<1|1, mid+1, r, val);
        return rightMost(pos<<1, l, mid, val);
    }
    bool MinusQuery(int q, int n) {                             // Decreases q nodes by 1
        if(q > n) return 0;
        int posVal = findVal(1, 1, n, q);
        if(posVal <= 0) return 0;
        int r = rightMost(1, 1, n, posVal);
        int l = rightMost(1, 1, n, posVal+1);
        int rem = q - l;
        if(l >= 1)          update(1, 1, n, 1, l, -1);
        if(r-rem+1 <= r)    update(1, 1, n, r-rem+1, r, -1);
        return 1;
}};

// Sqrt Decomposition
// Problem: https://www.codechef.com/problems/CHEFEXQ

// Operations:
// 1 : Update value x at pos i
// 2 : Find subarray XOR of value k from index 1 to r (All Subarray starts from 1)
// Approach:
// 1 : All segment consecutive xor is calculated in Seg aray
//   : All segment consecutive xor is also counted on SegMap
// 2 : Updates are done on each Decomposed segment array
// 3 : Queries are combined from all Decomposed array in the range

//---------------------------------- Sqrt Decompose Functions Start-----------------------------------//

int BlockSize, Seg[1010][1010];                  // BlockSize is the size of each Block
int SegMap[330][1110007] = {0};

void Update(int v[], int l, int val) {           // Updates value in position l : val
    int idx = l/BlockSize;                           // Block Index
    int lft = (l/BlockSize)*BlockSize;               // The leftmost index of array v, which is the 0 position of Segment idx
    v[l] = val;                                      // Setting value to default array to ease

    // Clear full block and re-calculate            // Using memset in large array will cause TLE
    SegMap[idx][Seg[idx][0]]--;                     // Decreasing previous value
    Seg[idx][0] = v[lft];
    SegMap[idx][v[lft++]]++;                        // Increasing with new value
    for(int i = 1; i < BlockSize; ++i, ++lft) {
        SegMap[idx][Seg[idx][i]]--;
        Seg[idx][i] = Seg[idx][i-1] ^ v[lft];
        SegMap[idx][Seg[idx][i]]++;
}}

int Query(int l, int r, int k) {                // Query in range l -- r for k
    int Count = 0, val = 0;
    while(l%BlockSize != 0 && l < r) {          // if l partially lies inside of a sqrt segment
        //cout << "P1" << endl;
        Count += (Seg[l/BlockSize][l%BlockSize] == k);
        val = val^Seg[l/BlockSize][l%BlockSize];
        ++l;
    }
    while(l+BlockSize <= r) {                   // for all full sqrt segment
        Count += SegMap[l/BlockSize][k^val];
        val ^= Seg[l/BlockSize][BlockSize-1];
        l += BlockSize;
    }
    while(l <= r) {                             // for the rightmost partial sqrt segment values
        Count += (Seg[l/BlockSize][l%BlockSize] == (k^val));
        ++l;
    }

    return Count;
}
void SqrtDecompose(int v[], int len) {           // Builds Sqrt segments
    int idx, pos, val = 0;
    BlockSize = sqrt(len);                       // Calculating Block size
    for(int i = 0; i < len; ++i) {
        idx = i/BlockSize;                       // Index of block
        pos = i%BlockSize;                      // Index of block element
        if(pos == 0) val = 0;
        val ^= v[i];
        Seg[idx][pos] = val;
        SegMap[idx][val]++;
}}
//----------------------------------- Sqrt Decompose Functions End----------------------------------//

int v[100100];
int main() {
    int n, q, idx, x, t;
    sf("%d %d", &n, &q);
    for(int i = 0; i < n; ++i)
        sf("%d", &v[i]);
    SqrtDecompose(v, n);
    while(q--) {
        sf("%d", &t);
        if(t == 1) {
            sf("%d %d", &idx, &x);
            Update(v, idx-1, x);
        }
        else {
            sf("%d %d", &idx, &x);
            pf("%d\n", Query(0, idx-1, x));
}}}

//Single Source Shortest Path (Negative Cycle)
//Complexity : O(VE)


vector<int>G[MAX], W[MAX];
int V, E, dist[MAX];
void bellmanFord(int source) {              // If there exists disconnected graphs, then add a dummy source node which will
    for(int i = 0; i <= V; i++)             // direct to all nodes with cost 0, and run bellmanFord from that virtual node
        dist[i] = INF;                      // set to -INF if max distance is needed
    dist[source] = 0;
    for(int i = 0; i < V-1; i++)                            // relax all edges V-1 times, if virtual node added, run V times
        for(int u = 0; u < V; u++)                          // all the nodes, if virtual node added, run within u <= V
            for(int j = 0; j < (int)G[u].size(); j++) {
                int v = G[u][j], w = W[u][j];               // relax edges, set to max if max value needed
                if(dist[u] != INF)                          // if there is a negative weight, then INF + negative weight < INF and INF becomes +-INF
                    dist[v] = min(dist[v], dist[u]+w);
}}

bool hasNegativeCycle() {
    for(int u = 0; u < V; u++)
        for(int i = 0; i < G[u].size(); i++) {      // if bellmanFord is run for max values, then this code will
            int v = G[u][i], w = W[u][i];           // return true for positive cycle by adding this line
            if(dist[v] > dist[u] + w)               // if(dist[v] < dist[u] + w)
                return 1;
        }
    return 0;
}

bool vis[MAX][2];
void negativePoint(int u) {                         // Works in undirected graph
    queue<pair<int, bool> >q;                       // if vis[v][1] == 1 then there exists an negative cycle
    q.push(make_pair(u, 0));                        // vis[v]][1] is true for all nodes which are in negative cycle and
    memset(vis, 0, sizeof vis);                     // the nodes that can be reached from the negative cycle nodes
    vis[u][0] = 1;                                  // on one/more path from u to v
    while(!q.empty()) {
        u = q.front().first;
        bool neg = q.front().second;
        q.pop();
        for(int i = 0; i < (int)G[u].size(); ++i) {
            int v = G[u][i];
            int w = W[u][i];
            if(dist[v] > dist[u] + w)
                neg = 1;
            if(vis[v][neg])
                continue;
            vis[v][neg] = 1;
            q.push(make_pair(v, neg));
}}}

//Strongly Connected Component (Tarjan)
//Complexity : O(V+E)

vector<int>G[MAX], SCC;
int dfs_num[MAX], dfs_low[MAX], dfsCounter, SCC_no = 0;
bitset<MAX>visited;
map<int, int>Component;         // For Creating new SCC (ConnectNode function)

void tarjanSSC(int u) {
    // Stack, here, it is implemented as vector instead
    SCC.push_back(u);
    // Marking node u as visited
    // visited[u] marks if the node u is usable in a SCC and not used on other SCC
    // if visited[u] is false, then it is used in other SCC
    visited[u] = 1;
    dfs_num[u] = dfs_low[u] = ++dfsCounter;
    // for all Strongly Connected Component (directed graph), dfs_low[u] is same
    for(int i = 0; i < (int)G[u].size(); i++) {
        int v = G[u][i];
        // if it is not visited yet, backtrack it
        if(dfs_num[v] == 0)
            tarjanSSC(v);

        // visited[v] is used to check if this node is not in any other SCC
        if(visited[v])
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
    }

    // in a SCC the first node of the SCC, node u is the first node in a SCC if dfs_low[u]
== dfs_low[v]
    // as we implementing stack like data structure, the nodes from top to u are on the same
SCC
    if(dfs_low[u] == dfs_num[u]) {
        SCC_no++;       // Component Node no. starts from 0

        // ------------------ Use if ONLY IF ConnectNode / Printing needed ----------------
---
        bool first = 1;
        while(1) {
            int v = SCC.back();
            SCC.pop_back();

            // node v is used, so marking it as false, so that the ancestor nodes
            // doesn't use this node to update it's value

            visited[v] = 0;
            // printf("%d\n", v);
            Component[v] = SCC_no;      // Marking SCC nodes to as same component
            if(u == v)
                break;
        }
        // printf("\n");
    }
}

void ConnectNode() {                    // This function can convert Components to a new graph (G1)
    map<int, int> :: iterator it = Component.begin();

    for( ; it != Component.end(); ++it) {
        for(int i = 0; i < (int)G[it->first].size(); ++i) {
            int v = G[it->first][i];
            if(it->second == Component[v])              // No Self loop in new graph
                continue;
            G1[it->second].push_back(Component[v]);
}}}


void RunSCC(int V) {
    memset(dfs_num, 0, sizeof(dfs_num));
    dfsCounter = 0;
    visited.reset();
    SCC_no = 0;
    for(int i = 1; i <= V; i++)
        if(dfs_num[i] == 0)
            tarjanSSC(i);
}

// Fast IO with Templates

#include <bits/stdc++.h>
using namespace std;
#define MAX                 510000
#define EPS                 1e-9
#define INF                 1e7
#define MOD                 1000000007
#define pb                  push_back
#define mp                  make_pair
#define fi                  first
#define se                  second
#define pi                  acos(-1)
#define pf                  printf
#define sf(XX)              scanf("%lld", &XX)
#define SIZE(a)             ((ll)a.size())
#define ALL(S)              S.begin(), S.end()
#define Equal(a, b)         (abs(a-b) < EPS)
#define Greater(a, b)       (a >= (b+EPS))
#define GreaterEqual(a, b)  (a > (b-EPS))
#define FOR(i, a, b)        for(register int i = (a); i < (int)(b); ++i)
#define FORR(i, a, b)       for(register int i = (a); i > (int)(b); --i)
#define FastIO              ios_base::sync_with_stdio(false); cin.tie(NULL);
#define FileRead(S)         freopen(S, "r", stdin);
#define FileWrite(S)        freopen(S, "w", stdout);
#define Unique(X)           X.erase(unique(X.begin(), X.end()), X.end())
#define STOLL(X)            stoll(X, 0, 0)

```

```
#define isOn(S, j)          (S & (1 << j))
#define setBit(S, j)        (S |= (1 << j))
#define clearBit(S, j)      (S &= ~(1 << j))
#define toggleBit(S, j)     (S ^= (1 << j))
#define lowBit(S)           (S & (-S))
#define setAll(S, n)        (S = (1 << n) - 1)

typedef unsigned long long ull;
typedef long long ll;
typedef map<int, int> mii;
typedef map<ll, ll>mll;
typedef map<string, int> msi;
typedef vector<int> vi;
typedef vector<ll>vl;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef vector<pair<int, int> > vii;
typedef vector<pair<ll, ll> >vll;

//int dx[] = {-1, 0, 1, 0}, dy[] = {0, 1, 0, -1};
//int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1}, dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};
//---------------------------------------------------------------------------------------------
----------------

inline void fastIn(int &num) {          // Fast IO, with space and new line ignoring
    bool neg = false;
    register int c;
    num = 0;
    c = getchar_unlocked();
    for( ; c != '-' && (c < '0' || c > '9'); c = getchar_unlocked());
    if (c == '-') {
        neg = true;
        c = getchar_unlocked();
    }
    for(; (c>47 && c<58); c=getchar_unlocked())
        num = (num<<1) + (num<<3) + c - 48;
    if(neg)
        num *= -1;
}

inline void fastOut (long long n) {
    long long N = n, rev, count = 0;
    rev = N;
    if (N == 0) { putchar('0'); return ;}
    while ((rev % 10) == 0) { count++; rev /= 10;}                  //obtain the count of
the number of 0s
    rev = 0;
    while (N != 0) { rev = (rev<<3) + (rev<<1) + N % 10; N /= 10;}  //store reverse of N in
rev
    while (rev != 0) { putchar(rev % 10 + '0'); rev /= 10;}
    while (count--) putchar('0');
}


// Scanf Trick
// input: (alpha+omega)^2
// scanf(" %*[(] %[^+] %*[+] %[^)] %s", a, b, n);
// %* is used for skipping
// %*[(] skipping (
// %[^+] take input until +
// %*[+] skipping +
// %*[^)] skipping ^ and )

// Tree Max Distance Node
// Set any node as root, then do dfs and find the farthest node, then again from that farthest node
// do dfs for farthest node, the two nodes are the farthest node

pii dfs(int u, int par, int d) {
    pii ret(d, u);                                      // {distance, node}
    for(int i = 0; i < (int)G[u].size(); ++i)
        if(G[u][i] != par)
            ret = max(ret, dfs(G[u][i], u, d+W[u][i]));
    return ret;
}

int GetDistance() {
    pii left = dfs(0, -1, 0);
    pii right = dfs(left.second, -1, 0);
    return right.first;
}

// Codeforces :E. Propagating tree ( http://codeforces.com/contest/384/problem/E )
// Given a tree (node 1 - n)
// perform two operations:
// 1. Add x value to node u, Add -x value to node u's immediate children, Add x to their immediate children, and so on
// in other words, add value x to all childs where (parentLevel%2 == childLevel%2), add -val otherwise
// 2. Output value of node u

vector<int> G[MAX];
int sTime[MAX], eTime[MAX], level[MAX], cst[MAX], timer;
BIT EvenNode, OddNode;

/* sTime : starting time of node n
   eTime : finishing time of node n
     1
    / \
   5   6
      / \
     7   4
        / \
       2   3
discover nodes : {1, 5, 6, 7, 4, 2, 3}
sTime[] = {1, 6, 7, 5, 2, 3, 4}    index starts from 1, i'th index contains start time of i'th node
eTime[] = {7, 6, 7, 7, 2, 7, 4}

calculate child :
for node 6 : childs are in range sTime[6] - eTime[6] : 3 - 7
so child nodes are : 6, 7, 4, 2, 3 (discover node index range)
we don't need discover time vector to calculate distance
notice, if we only update with sTime and eTime, the range update will always be right */

void dfs(int u, int lvl) {
    sTime[u] = ++timer;
    level[u] = lvl;
    for(int i = 0; i < (int)G[u].size(); ++i)
        if(sTime[G[u][i]] == 0)
            dfs(G[u][i], lvl+1);
    eTime[u] = timer;
}

void AddVal(int node, int val) {
    if(level[node]%2 == 0) {
        EvenNode.update(sTime[node], eTime[node], val);
        OddNode.update(sTime[node], eTime[node], -val);
    }
    else {
        EvenNode.update(sTime[node], eTime[node], -val);
        OddNode.update(sTime[node], eTime[node], val);
}}

int GetVal(int node) {                                      // cst[node] contains initial cost (if exists)
    return cst[node] + (level[node]%2==0 ? EvenNode.read(sTime[node]):OddNode.read(sTime[node]));
}


// Complete Binary Tree
// Sum of distance from a node "n" such that every nodes distance from node "n" is less than or equal to k
// http://mishadoff.com/blog/dfs-on-binary-tree-array/

vector<ll>v[MAX], w, sum[MAX];          // W[i] contains weight of I'th node
int n, m;
void dfs(int node = 1) {                // node starts from 1
    if(node > n) return;

    ll lft = node<<1, rht = node<<1|1;
    dfs(lft), dfs(rht);
    ll lftSize = v[lft].size(), rhtSize = v[rht].size();
    ll nodeSize = lftSize+rhtSize+1;
    v[node].resize(nodeSize);
    v[node][0] = 0;                             // distance from this node to this node

    //printf("node : %d, nodeSize : %d, lftSize : %d, rhtSize : %d\n", node, nodeSize, lftSize, rhtSize);
    ll l = 0, r = 0;
    for(ll i = 1; i < nodeSize; ++i) {
        if(l == lftSize)
            v[node][i] = v[rht][r++] + w[rht];
        else if(r == rhtSize)
            v[node][i] = v[lft][l++] + w[lft];
        else {
            int lftW = v[lft][l] + w[lft], rhtW = v[rht][r] + w[rht];
            if(lftW < rhtW) {
                v[node][i] = lftW;
                l++;
            }
            else {
                v[node][i] = rhtW;
                r++;
}}}}

ll single(int node, ll d, ll delta) {
    if(d < 0) return 0;
    ll n = upper_bound(v[node].begin(), v[node].end(), d) - v[node].begin();
    return sum[node][n-1] + delta*n;                                    // delta is the common distance of all nodes
}

ll query(int node, ll k) {
    ll ans = single(node, k, 0);
    ll totlen = 0;
    while(node/2) {
        totlen += w[node];
        ll tmp = single(node/2, k-totlen, totlen);                      // distances from parent node
        tmp -= single(node, k-totlen-w[node], totlen + w[node]);        // common overlapped distance (of child node) from parent node
        ans += tmp;
        node /= 2;
    }
    return ans;
}

void PreCal() {                             // First run dfs(), then run PreCal()
    for(int i = 1; i <= n; ++i) {
        sum[i].resize(v[i].size());
        sum[i][0] = v[i][0];
        for(int j = 1; j < SIZE(v[i]); ++j)
            sum[i][j] = sum[i][j-1] + v[i][j];
}}

//Trie
//Complexity : making a trie : O(S), searching : O(S)

struct node {
    bool isEnd;
    node *next[11];
    node() {
        isEnd = false;
        for(int i = 0; i < 10; i++)
            next[i] = NULL;
}};

//trie of a string abc, ax
// [start] --> [a] --> [b] --> [c] --> endMark
//                |
//               [x] --> endMark

//creates trie, returns true if the trie we are creating is a segment of a string
//to only create a trie remove lines which are comment marked

bool create(char str[], int len, node *current) {
    for(int i = 0; i < len; i++) {
        int pos = str[i] - '0';
        if(current->next[pos] == NULL)
            current->next[pos] = new node();
        current = current->next[pos];
        if(current->isEnd)  //
            return true;    //
    }
    current->isEnd = true;  //
    return false;           //
}

void del(node *current) {
    for(int i = 0; i < 10; i++)
        if(current->next[i] != NULL) del(current->next[i]);
    delete current;
}

void check(node *current) {
    for(int i = 0; i < 10; i++) {
        if(current->next[i] != NULL)
            check(current->next[i]);
    }
    if(found) return;
    if(current->isEnd && !found) {
        for(int i = 0; i < 10 && !found; i++)
            if(current->next[i] != NULL) {
                found = 1;
}}}

int main() {
    //freopen("in", "r", stdin);
    //freopen("out", "w", stdout);
    int t, n;
    char S[15];
    scanf("%d", &t);
    while(t--) {
        found = 0;
        node* root = new node();        //important part of the code
        scanf("%d", &n);
        while(n--) {
            scanf(" %s", S);
            if(!found)
                if(create(S, strlen(S), root))
                    found = 1;
        }
        if(!found) check(root);
        if(found)  printf("NO\n");
        else       printf("YES\n");
        del(root);
}}
```