```
/* -------------------------------------CPP Templates -------------------------------------- */
#define MAX                  510000
#define EPS                  1e-9
#define INF                  0x3f3f3f3f
#define MOD                  1000000007
#define pi                   acos(-1)
#define Equal(a, b)          (abs(a-b) < EPS)
#define Greater(a, b)        (a >= (b+EPS))
#define GreaterEqual(a, b)   (a > (b-EPS))
#define FastIO               ios_base::sync_with_stdio(false); cin.tie(NULL);
#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;}  // 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 ) */
```

```

/* ----------------------------------- Data Structure ----------------------------------- */
// Policy Based Data structure
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#define at(X)          find_by_order(X)
#define lessThan(X)    order_of_key(X)
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, updating node policy
   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>
   find_by_order(x) returns x'th elements iterator
   order_of_key(x) returns number of values less than (or equal to) x */

/* ------------------------------ DSU with Path Compression ------------------------------ */
struct DSU {
    vector<int>u_list, u_set;                       // u_list[x] : the size of a set x
    DSU() {}                                        // u_set[x]  : the root of x
    DSU(int SZ) { init(SZ); }
    int unionRoot(int n) {                          // Root of node n
        if(u_set[n] == n) return n;
        return u_set[n] = unionRoot(u_set[n]); }
    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;
    }
    bool isRoot(int x)              { return u_set[x] == x; }
    bool isRootContainsMany(int x)  { return (isRoot(x) && (u_list[x] > 1)); }
    bool isSameSet(int a, int b)    { return (unionRoot(a) == unionRoot(b));
}};
// Bipartite DSU (Tested)
struct BipartiteDSU {
    vector<int>u_list, u_set, u_color;                      // u_color : color of nodes
    vector<bool>missmatch;                                  // Bicolor missmatch
    BipartiteDSU() {}
    BipartiteDSU(int SZ) { init(SZ); }
    pll unionRoot(int n) {                                  // Finds root of node n
```

```
        if(u_set[n] == n) return {n, u_color[n]};           // returns : {root_node, color}
        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;
        return {u_set[n], u_color[n]};
    } int makeUnion(int a, int b) {
        int x = unionRoot(a).first, y = unionRoot(b).first;
        if(x == y) {
            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 y root
            u_set[x] = y, u_list[x] += u_list[y];
            u_color[x] = (u_color[a]+u_color[b]+1)&1;      // Setting color of component
            return y;                                      // y according to the color of a & b
        } else {                                           // Makes x root
            u_set[y] = x, u_list[y] += u_list[x];
            u_color[y] = (u_color[a]+u_color[b]+1)&1;
            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;
}};
// Dynamic Weighted DSU  ------ Not Tested !!!!!!!!!!!!
struct WeightedDSU {
    vector<int>u_list, u_set, u_weight, weight;
    WeightedDSU() {}
    WeightedDSU(int SZ) { init(SZ); }
    int unionRoot(int n) {
        if(u_set[n] == n) return n;
        return u_set[n] = unionRoot(u_set[n]);
    } void changeWeight(int u, int w, bool first = 1) {        // Change any component's weight
        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]) {
            u_set[y] = x, u_list[x] += u_list[y];
            u_weight[x] += u_weight[y];
            return x;
        } else {
            u_set[x] = y, u_list[y] += u_list[x];
            u_weight[y] += u_weight[x];
            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;
}};
/* ------------------------------------- Trie -------------------------------------
 Complexity : making a trie : O(S), searching : O(S)

    Trie of a string abca, abcb:
                                               (b)    {isEnd = 1}
                                            |----------> node5
   [start] ----> node1 ----> node2 ----> node3 ----->  node4
           (a)         (b)         (c)         (a)   {isEnd = 1}
    Edges are the next[x] pointers, that direct to the next node of the trie */
// Dynamic Trie (with pointers)
struct dynamicTrie {
    struct node {
        bool isEnd;
        node *next[CHARS];                                      // CHARS: number of charachters
        node() {
            isEnd = false;
            for(int i = 0; i < 10; i++) next[i] = NULL;
    }};
    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 < CHARS; i++)
            if(current->next[i] != NULL)
                del(current->next[i]);
        delete current;
    } void check(node *current) {
        for(int i = 0; i < CHARS; i++)
            if(current->next[i] != NULL)
                check(current->next[i]);
        if(found) return;
        if(current->isEnd && !found) {
            for(int i = 0; i < CHARS && !found; i++)
                if(current->next[i] != NULL) {
                    found = 1;
}}}};
// Non-Dynamic implementation
// root node is at 0 index of tree
// root node counter contains total number of string insertion
// each inserted char counter is on the child node of the edges
struct Trie {
    struct node {
        int cnt;                    // number of edges, or number of times this node is visited
        int nxt[CHARS];                // if nxt[x] == -1, there is no edge from this node to x
    };
```

```
    int nodes;
    node tree[MAX];
    void newNode() {
        tree[nodes].cnt = 0;
        memset(tree[nodes].nxt, -1, sizeof tree[nodes].nxt);
        ++nodes;
    } void init() { nodes = 0; newNode(); }
    int getId(char x) {
        if(x >= 'A' and  x <= 'Z')
            return (x - 'A' + 27);
        return (x - 'a' + 1);
    } void insert(string &str, int len = 0, int idx = 0) {
        tree[idx].cnt++;
        if(len == str.size()) return;
        int id = getId(str[len]);
        if(tree[idx].nxt[id] == -1) {
            tree[idx].nxt[id] = nodes;
            newNode();
        } insert(str, len+1, tree[idx].nxt[id]);
    } int search(string &str, int len = 0, int idx = 0) {
        if(len == str.size())       return -2;
        int id = getId(str[len]);
        if(tree[idx].nxt[id] == -1) return -1;
        if(tree[idx].cnt == 1)      return len;
        return search(str, len+1, tree[idx].nxt[id]);
}};

/* ---------------------------------- Sqrt Decompose ---------------------------------- */
int BlockSize, seg[1010];                        // BlockSize is the size of each Block
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
        /* perform operation */
}} void Update(int v[], int l, int val) {           // Updates value in position l : val
    int idx = l/BlockSize;                          // Block Index
    int startPos = (l/BlockSize)*BlockSize;         // The starting position from where
    for(int i = 1; i < BlockSize; ++i) {            // the block contains value of the input
        seg[idx][i] = v[startPos++];                // array v[]
}} int Query(int l, int r) {                        // Query in range l -- r
    int Count = 0, val = 0;
    while(l%BlockSize != 0 && l < r) {             // l partially lies inside of a sqrt segment
        /* perform operation from the input array */
        ++l;
    } while(l+BlockSize <= r) {                       // for all full sqrt segment
        /* perform operation from seg[l/BlockSize] */
        l += BlockSize;
    } while(l <= r) {                                 // rightmost partial sqrt segment
        /* perform operation from the input array */
        ++l;
    } return Count;
}
```

```cpp
/* ------------------------------------- MO's on array -------------------------------------
 Complexity : (N+Q)*sqrt(N)*InsertEraseConstant + (Q*QueryProcessingConstant) */
struct query {
    int l, r, id;
} q[MAX];
const int block = 320;                  // For N = 100000, sqrt(N) = 320, always use as const
// MO's tree ordering with only query processing
bool cmp(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;           // Even-Odd sorting
    return a.r > b.r;
}                                                 // MOs might work fast for a larger block size
void add(int x) {}                                   // Add x'th value in range
void remove(int x) {}                                // Remove x'th value from range
void MOs() {
    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
}}
/* ----------------------------------- MO's on SubTree -----------------------------------
 Sort subtree parents according to l = in[u] and r = out[u], ID[timer] = node
 Iterate over the dfs timer and apply MOs in the [l, r] range, add/remove is same as above
*/
int timer = -1;
void dfs(int u, int p = 0) {                                          // MO's Sub-Tree DFS Timing
    in[u] = ++timer;
    ID[timer] = u;
    for(auto v : G[u])
        if(v != p) dfs(v, u);
    out[u] = timer;
}
/* ----------------------------------- MO's on Tree Path ----------------------------------
   Perform Query operation from path u to v, iterate over dfs-time */
struct query {
    int id, ut, vt, lca;        // timing of node u, node v and lca of (u & v)
} q[MAX];
int timer = -1;
void dfs(int u, int p = 0) {
    in[u] = ++timer;
    ID[timer] = u;
    for(auto v : G[u])
        if(v != p) dfs(v, u);
    out[u] = ++timer;
} bitset<MAX> proc;
void process(int ut) {      // ADD and REMOVE in same function
    if(proc[ID[ut]]) {}     // ADD:    if proc = 0, then add the node and set proc = 1
    else             {}     // REMOVE: else remove the node and set proc = 0
} void MOs_Tree() {
    for(int i = 0, j = 0; j < Q; ++i, ++j) {        // Input Processing
```

```
        scanf("%d%d", &u, &v);
        q[i].id = i, q[i].lca = LCA(u, v);
        if(in[u] > in[v])       swap(u, v);
        if(q[i].lca == u)       q[i].ut = in[u], q[i].vt = in[v];
        else                    q[i].ut = out[u], q[i].vt = in[v];
    } sort(q, q+Q, cmp);
    int l = 0, r = -1;
    for(int i = 0; i < Q; ++i) {
        while(l > q[i].ut)   proccess(--l);
        while(r < q[i].vt)   proccess(++r);
        while(l < q[i].ut)   proccess(l++);
        while(r > q[i].vt)   proccess(r--);
        u = ID[l], v = ID[r];
        if(q[i].lca != u and q[i].lca != v) proccess(in[q[i].lca]);
        ans[q[i].id] =      // Calculate the answer
        if(q[i].lca != u and q[i].lca != v) proccess(in[q[i].lca]);
}}

/* ------------------------------------ 1D Fenwick Tree ------------------------------------ */
struct BIT {
    ll tree[MAX], 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), haven't used often
        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)   pos = mid, hi = mid-1;
            else                    lo = mid+1;     // read(mid) >= cSum : lowest index of cSum
        } return pos;                              // read(mid) == cSum : highest index of cSum
    }
    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) { return read(MaxVal-idx); }            // gives summation from 1 to idx
    void invInsert(int idx) { update(MaxVal-idx, 1); }      // adds 1 to all index less than idx
    void invRemove(int idx) { update(MaxVal-idx, -1); }     // removes 1 from idx
    void invUpdate(int idx, ll val) { update(MaxVal-idx, val); }
};
/* -------------------------- 2D Fenwick Tree ------------------------
  /\|   (x1,y2) -------- (x2,y2)
   | |         |        |
   y |         ---------
     |   (x1,y1)         (x2, y1)
     |___________________________
(0, 0)                     x-->        */
struct FTree2D {
    ll tree[MAX][MAX] = {0};
    int xMax, yMax;
    void init(int xx, int yy) { xMax = xx, yMax = yy; }
    void update(int x, int y, int val) {
        for(int x1 = x; x1 <= xMax; x1 += x1 & -x1)
            for(int y1 = y; y1 <= yMax; y1 += y1 & -y)
                tree[x1][y1] += val;
    } ll read(int x, int y) {
        ll sum = 0;
        for(int x1 = x; x1 > 0; x1 -= x1 & -x1)
            for(int y1 = y; y1 > 0; y1 -= y1 & -y1)
                sum += tree[x1][y1];
        return sum;
    } ll readSingle(int x, int y) {
        return read(x, y) - read(x-1, y) - read(x, y-1) + read(x-1, y-1);
    } void updateSquare(int x1, int y1, int x2, int y2, int val) {
        update(x1, y1, val),    update(x1, y2+1, -val);
        update(x2+1, y1, -val), update(x2+1, y2+1, val);               // p1 : lower left point
    } ll readSquare(int x1, int y1, int x2, int y2) {                 // p2 : upper right point
        return  read(x2, y2) - read(x1-1, y2) - read(x2, y1-1) + read(x1-1, y1-1);
}};

/* -------------------------- 3D Fenwick Tree ------------------------ */
ll xMax = 100, yMax = 100, zMax = 100, tree[105][105][105];
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, 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);
}
// Pattens to built BIT update read: always starts with first(starting point), take (1 to n)
// elements from ending point with all combination add it to staring point, add (-1)^n * val
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);
}

/* -------------------------------------- Segment Tree ------------------------------------ */
// 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]; 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]);
    } 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));
        sum[pos<<1|1]  = (sum[pos<<1|1] + prop[pos]*(r-mid));
        prop[pos<<1]   = (prop[pos<<1] + prop[pos]);
        prop[pos<<1|1] = (prop[pos<<1|1] + prop[pos]);
        prop[pos]      = 0;
    } void update(int pos, int l, int r, int L, int R, ll val) {         // Range [L, R] Update
        if(r < L || R < l) return;
        propagate(pos, l, r);
        if(L <= l && r <= R) {
            sum[pos] = (sum[pos] + val*(r-l+1)), prop[pos] = (prop[pos] + 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);
        sum[pos] = (sum[pos<<1] + sum[pos<<1|1]);
    } ll query(int pos, int l, int r, int L, int R) {                    // Range [L, R] Query
```

```
        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));
}};
// Segment Tree Insert/Remove value, Find K'th Value
struct KthValueInsertErase {                     // Finds/Deletes K'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) {    // removeVal = 1
        if(L == R) { tree[pos] = (removeVal ? 0:1); return L; }            // removes the value
        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 [set, reset, flip]                  [Problem: UVA 11402 Ahoy Pirates]
struct RangeBitQuery {
    vector<pair<int, int> >tree;                       // number of set bits, propagation state
    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 parentState) {                 // Generates child state w.r.t parent's state
        if(parentState == 1) return 2;               // 2 : all set to zero
        if(parentState == 2) return 1;               // 1 : all set to one
        if(parentState == 3) return 0;               // 0 : no change
        return 3;                                    // 3 : all need to be flipped
    } void Propagate(int L, int R, int parent) {
        if(tree[parent].second == 0 or L == R) return;
        int mid = (L+R)>>1, 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(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;                             // Clear parent node prop state
```

```
    } 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) {          // Number of set bits [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) {                                                 // Query Part
            return (int)tree[pos].size() - (upper_bound(tree[pos].begin(), tree[pos].end(), k)
            - tree[pos].begin()); }
        int mid = (l+r)>>1;
        return query(pos<<1, l, mid, L, R, k) + query(pos<<1|1, mid+1, r, L, R, k);
}};
// Maximum cumulative sum from all possible range in a segment
struct RangeMaxSumNode {                // Range maximum sum node is coded
    ll sum, prefix, suffix, ans;        // seg sum, max prefix sum, max suffix sum, max sum
    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));
}};
// Bracket segment validity check and update single position bracket
struct BracketTreeNode {                // Only node merge is coded in note
    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;
}}};
// Outputs longest balanced bracket sequence in range [L, R]
struct node {
    ll lftBracket, rhtBracket, Max;
    node(ll lft=0, ll rht=0, ll Max=0) {      // Call: ( = node(1, 0, 0),   ) = node(0, 1, 0)
        this->lftBracket = lft;                     // number of left brackets
        this->rhtBracket = rht;                     // number of right brackets
        this->Max = Max;                            // The max len of bracket, output Max*2
    } void Merge(node lft, 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);
}};
// Path Compression
void CompressPath(vector<int> &point) {
    point.push_back(0);
    sort(point.begin(), point.end());
    point.erase(unique(point.begin()+1, point.end()), point.end());
}
// Offline Processing   [this code finds unique values in range l-r]
// The processing is done backwards, first we go to the right range r, then find ans in [l - r]
struct OfflineProcessing {
    int tree[4*MAX], v[MAX], IDX[MAX];      // IDX[x] keeps track of where x previously occured
    map<int, vector<int> > QueryEnd;            // Contains start positions for a end pos r
    map<pair<int, int>, int>Ans;                // Contains answer for ranges
    vector<pair<int, int> > Query;              // Contains query ranges
    void ArrayInput(int arraySize) {  for(int i = 1; i <= SZ; ++i) scanf("%d", &v[i]); }
    void QueryInput(int querySize) {
        int l, r;
        while(q--) {
            scanf("%d %d", &l, &r);
            Query.push_back(make_pair(l, r));
```

```
            QueryEnd[r].push_back(l);                        // Used for sorting
    }} void Process(int arraySize) {
        map<int, vi> :: iterator it;
        int lPos = 0;
        for(it = QueryEnd.begin(); it != QueryEnd.end(); ++it) {
            while(lPos < it->first) {
                lPos++;
                if(IDX[v[lPos]] == -1) { IDX[v[lPos]] = lPos, update(1, 1, SZ, lPos, 1); }
                else {
                    int pastIDX = IDX[v[lPos]];
                    IDX[v[lPos]] = lPos;
                    update(1, 1, SZ, pastIDX, -1);      // Remove count from past-left index
                    update(1, 1, SZ, lPos, 1);          // Add count to the latest index
            }}
            for(int i = 0; i < (int)(it->second).size(); ++i)
                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)]);
}};
/* ---------------------------- Persistant/Dynamic Segment Tree -----------------------------*/
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();
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;
} void update(node *pos, ll l, ll r, ll L, ll R, ll val) {                  // Range [L, R] update
    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), 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), y = query(pos->rht, mid+1, r, L, R);
    return x+y;
} void ClearTree(node *pos) {                      // Erasing A segment tree call: ClearTree(root)
    if(pos == NULL) { delete pos; return; }
```

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

/* ------------------------------ Dynamic Programming ------------------------------ */
// 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 left and right
// String Printer function of above DP
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]);
}}
// Checks if substring l-r is palindrome
bool isPalindrome(int l, int r) {
    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;
}
// Given two string s1 and s2, match the two strings by performing the following operations:
// delete chars, insert chars, and change chars at any position on any string
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;
}
// Printing the string of above dp function
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
}
// Reduce string AXDOODOO (len : 8) to AX(DO^2)^2 (len : 4)
int reduce(int l, int r) {
    if(l > r)           return INF;
    if(l == r)          return 1;
    if(dp[l][r] != -1)  return dp[l][r];
    int ret = r-l+1, len = r-l+1;
    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
   TAC + ACT + CTA = ACTAC */
int matchDP[20][20];
int TryMatch(int x, int y) {            // Finds overlap of two strings if placed as x + y
    if(matchDP[x][y] != -1)             // ABAAB + AAB : Match at index 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) {                 // Final match patterns of strings
    if(dp[last][mask] != -1)             return dp[last][mask];
    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 the final string
    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), 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));
}}
// FileName : 1141 - Brackets Sequence
// Given a bracket sequence of () and [] which can be non-accurate have to make it accurate
// such as the accurate sequence length is minimum and lexicographically smallest
int recur(int l, int r) {
    int &ret = dp[l][r];
    if(l > r)                     return ret = 0;
    if(l == r)                    return ret = 2;           // We need to place an extra bracket
    if(ret != INF)                return ret;
    ret = min(recur(l+1, r), recur(l, r-1))+2;      // First we assume that we
    char lft = s[l];                                // need to place brackets on first or on last
    if(lft == '(' or lft == '[') {                  // If this segment starts with opening bracket
      for(int i = l+1; i <= r; ++i) {         // Then we try to slice the segment into two parts
        if((lft == '(' and s[i] == ')') or (lft == '[' and s[i] == ']'))
          ret = min(ret, recur(l+1, i-1)+recur(i+1, r)+2);      // +2 is the lenght of () or []
return ret; }
/* -----------------------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)
MaxDigit contains values in reverse order, (123 will be stored as {3, 2, 1}) */
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, lim = (tight)? MaxDigit[idx] : 9;
    for(int i = 0; i <= lim; i++) {
        bool newTight = (MaxDigit[idx] == i)? tight:0;                  // caclulating newTight
```

```
        ll newValue = value ? ((value*10) % mod)+i : i;                 // value for next state
        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 in range [0 - N]
pairs are number of paired bits, prevOn shows if previous bit was on (it is for this problem)*/
ll dp[33][33][2][2], N, lastBit;
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);
    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; }

/* ----------------------- Double Bounded Digit Dp Technique -----------------------
mn, mx contains the digits from MSB to LSB and both of them must be of same length */
vector<int>tt, mn = {0, 5, 4}, mx = {1, 3, 0};              // mn = 54, mx = 130, mn is resized
void recur(int pos = 0, bool lower = 1, bool higher = 1) {                  // A dummy function
    if(pos == LEN) {
        for(auto it : tt) cout << it;
        cout << endl;
        return;
    } int lo = lower ? mn[pos]:0, hi = higher ? mx[pos]:9;
    for(int d = lo; d <= hi; ++d) {
        bool newLower = (d == mn[pos]) ? lower:0, newHigher = (d == mx[pos]) ? higher:0;
        tt.push_back(d);
        recur(pos+1, newLower, newHigher);
        tt.pop_back();
}}
/* 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[iteration_state][sum_difference] = maximum sum
    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
}
/* 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) */
int n, x[11], y[11], dist[11][11], memo[11][1 << 11], dp[11][1 << 11];
int TSP(int u, int bitmask) {               // TSP(startin_node, bitmask_of_visited_node)
    if(bitmask == ((1 << (n)) - 1))         return dist[u][0];
    if(dp[u][bitmask] != -1)                return dp[u][bitmask];
    int ans = INF;
    for(int v = 0; v <= n; v++)                 // Traverse all nodes from u
        if(u != v && !(bitmask & (1 << v)))
            ans = min(ans, dist[u][v] + tsp(v, bitmask | (1 << v)));
    return dp[u][bitmask] = ans;
}

/* ---------------------------------------- Graph ---------------------------------------- */
/* -------------------------------- 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)       Cycle++;        // If any Parent found (BackEdge)
        else if(color[v] == 0)  dfs(v);
    } color[u] = 1;                             // Mark as visited
}
/* ------------------------------- 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) {
    queue<int>Q;
    vector<int>dist(vertices+5, INF);                                 // distance vector
    Q.push(source_node), parent.resize(vertices+5, -1);   // parent vector is for path printing
    dist[source_node] = 0;
    while(not Q.empty()) {
        int u = Q.front();
        Q.pop();
        if(u == finish_node)  return dist[u];             // remove this line if shortest path
        for(int i = 0; i < G[u].size(); i++) {                      // to all nodes are needed
            int v = G[u][i];
            if(dist[v] == INF) {
                dist[v] = dist[u] + 1;
                parent[v] = u, Q.push(v);
    }}} return -1;
}
/* ------------------------------- Bi-coloring using BFS ------------------------------- */
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(auto v : G[u]) {
            if(color[v] == -1) {
                color[u] = (color[u] == 1) ? 2:1, q.push(v);
}}}}
/* ------------------------------ Shortest Path (Dikjstra) ------------------------------
   Complexity : (V*logV + E) */
vector<int>dist, G[MAX], W[MAX];                    // distance, edge list, weight list
void dikjstra(int u, int destination, int nodes) {
    dist.resize(nodes+1, INF);
    dist[u] = 0;
    priority_queue<pair<int, int> > pq;
    pq.push({0, -u});
    while(not pq.empty()) {
        int u = -pq.top().second, wu = -pq.top().first;     // node, weight sum
        pq.pop();
        if(u == destination) return;                        // if destination found, return
        if(wu > dist[u]) continue;                          // if weight is heavy, skip
        for(int i = 0; i < G[u].size(); i++) {
            int v = G[u][i], wv = W[u][i];                  // node, weight
            if(wu + wv < dist[v]) {                         // path relax
                dist[v] = wu + wv;
                p[v] = u;                                   // for path printing
                pq.push({-dist[v], -v});
}}}} void printPath(int u) {                                // path print for dikjstra
    if (u == s) {   printf("%d", s); return; }
    printPath(p[u]);                        // recursive: to make the output format: s -> ... -> t
    printf(" %d", u);
}
/* 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({0, Start});
    while(!pq.empty()) {
        int u = pq.top().second, 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)        dist[u].push_back(w);
        if((int)dist[u].size() > Kth)       continue;
        for(int i = 0; i < (int)G[u].size(); ++i) {
            int v = G[u][i], _w = w + W[u][i];
            if((int)dist[v].size() == Kth) continue;
            pq.push(make_pair(-_w, v));
    }} return -1;
}
/* Kth Shortest Path (Visits Same Edge More Than Once if required) */
int KthDikjstra(int Start, int End, int Kth) {      //
```

```
    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, w = -pq.top().first;
        pq.pop();
        if(dist[u].empty()) dist[u].push_back(w);
        else if(dist[u].back() != w) {
            if((int)dist[u].size() < Kth)      dist[u].push_back(w);
            else if(dist[u].back() <= w)       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], _w = w + W[u][i];
             pq.push(make_pair(-_w, v));
    }}
    if((int)dist[End].size() < Kth) return -1;
    return dist[End].back();
}
/* -------------------- Single Source Shortest Path (Negative Cycle) -------------------------
   Complexity : O(VE) */
vector<int>G[MAX], W[MAX];
int V, E, dist[MAX];
void bellmanFord(int source) {
    for(int i = 0; i <= V; i++) dist[i] = INF;
    dist[source] = 0;
    for(int i = 0; i < V-1; i++)                                   // relax all edges V-1 times
        for(int u = 0; u < V; u++)
            for(int j = 0; j < (int)G[u].size(); j++) {
                int v = G[u][j], w = W[u][j];
                if(dist[u] != INF)
                    dist[v] = min(dist[v], dist[u]+w);
}} bool hasNegativeCycle() {                           // if bellmanFord is run for max values,
    for(int u = 0; u < V; u++)                         // then this code will return true for
        for(int i = 0; i < G[u].size(); i++) {         // positive cycle by adding this line
            int v = G[u][i], w = W[u][i];              // 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
    memset(vis, 0, sizeof vis);       // in negative cycle and the nodes that can be reached
    vis[u][0] = 1;                    // from the negative cycle nodes on one/more
    while(!q.empty()) {               // path from u to v
        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], 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));
}}}
/* ------------------------ All Pair Shortest Path - Floyd Warshal ---------------------------
   Complexity : O(V^3) */
int G[MAX][MAX], parent[MAX][MAX];
void graphINIT() {
    memset(G, INF, sizeof G);
    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 and all combination of
        for(int i = 0; i < V; i++)                          // source(i) and destination(j)
            for(int j = 0; j < V; j++)
                if(G[i][k] != INF && G[k][j] != INF) {          // if G[i][i] = negative, then
                    G[i][j] = min(G[i][j], G[i][k]+G[k][j]);    // 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) {                                     // maximum weight of minimum cost path
    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) {                    // Checks if there exists a path from i to j
    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]);
}
/* --------------------- MST Kruskal + Union Find Disjoint Set (DSU) -------------------------
   Complexity of MST : O(E logV) */
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, v = (*it).second.second, 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], 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
        if(!taken[v]) mst_cost += w, process(v);             // as it contains the minimum edge
    } printf("Prim's MST cost : %d\n", mst_cost);
    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, cur_nodes = V;
    while(true) {                            // lo[v] : contains minimum weight to go to node v
        vector<int> lo(cur_nodes, INF), pi(cur_nodes, INF);
        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;                          // 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])
                mark[u] = i;
            if(u != root && id[u] < 0) {
                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;                              // all nodes are 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
}
/* ------------------------------ Articulation Point ------------------------------
   Complexity O(V+E)
   Tarjan, DFS Timing
   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 */
vector<int>G[101];
int dfs_num[101], dfs_low[101], parent[101], isAtriculationPoint[101];
int dfsCounter, rootChildren, dfsRoot;
void articulationPoint(int u) {
    dfs_low[u] = dfs_num[u] = ++dfsCounter;
    for(auto v : G[u]) {
        if(dfs_num[v] == 0) {
            parent[v] = u;                               // Special case for root node, if root
            if(u == dfsRoot) rootChildren++;             // node has child, increment counter
            articulationPoint(v);
            if(dfs_num[u] <= dfs_low[v] && u != dfsRoot)                     // Avoiding root node
                isArticulationPoint[u]++;
            dfs_low[u] = min(dfs_low[v], dfs_low[u]);
        } else if(parent[u] != v)
            dfs_low[u] = min(dfs_low[u], dfs_num[v]);
}} int main() {
    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);
    }}                        // isAtriculationPoint + 1 = number of nodes that is disconnected
    for(int i = 0; i < 101; i++)                                // Printing Articulation Points
        if(isArticulationPoint[i]) printf("%d ", i);
    printf("%d\n", (int)isArticulationPoint.count());
}
/* ---------------------------------------- Bridge ----------------------------------------
   Complexity : O(V+E) */
vector<int> G[MAX];
vector<pair<int, int> >ans;
int dfs_num[MAX], dfs_low[MAX], dfsCounter, timeToNode[MAX];
void bridge(int u, int par = -1) {
    dfs_num[u] = dfs_low[u] = ++dfsCounter;
    timeToNode[dfs_num[u]] = u;                     // For building new tree from current graph
    for(auto v : G[u]) {
        if(v == par) continue;
        if(dfs_num[v] == 0) {
            bridge(v, u);
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
            if(dfs_num[u] < dfs_low[v])
                ans.push_back(make_pair(min(u, v), max(u, v)));
        } else if(v != par) dfs_low[u] = min(dfs_low[u], dfs_num[v]);
    }
    timeToNode[dfs_num[u]] = u;                  // If BuildNewTree is used otherwise ignore it
```

```
} 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);
}
// Make tree from the above found connected components
vi Tree[MAX];
int conv[MAX] = {0}, ncnt;
int Convert(int u) {                                  // Converts graph node number to
    if(conv[dfs_low[u]] == 0)                         // tree numbers
        conv[dfs_low[u]] = ++ncnt;                    // tree nodes start from 1
    return conv[dfs_low[u]];                               // ncnt contains total number of nodes
} int findMin(int u) {                            // Basic tarjan doesn't contain same dfs_low[u]
    if(dfs_low[u] == dfs_num[u]) return dfs_low[u];                            // for all nodes,
    return dfs_low[u] = findMin(timeToNode[dfs_low[u]]);     // so this finds the actual values
} int BuildNewTree(int V) {
    ncnt = 0;
    for(int i = 1; i <= V; ++i)  findMin(i);
    for(auto it : ans) {
        int u = Convert(it.first), v = Convert(it.second);
        Tree[u].pb(v),Tree[v].pb(u);
    } return ncnt;
}
/* -------------------------- 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) {
    SCC.push_back(u), visited[u] = 1, dfs_num[u] = dfs_low[u] = ++dfsCounter;
    for(auto v : G[u]) {
        if(dfs_num[v] == 0)     tarjanSSC(v);
        if(visited[v])          dfs_low[u] = min(dfs_low[u], dfs_low[v]);
    } if(dfs_low[u] == dfs_num[u]) {
        SCC_no++;                                            // Component Node no. starts from 0
        bool first = 1;
        while(1) {
            int v = SCC.back();
            SCC.pop_back(), visited[v] = 0;
            // printf("%d\n", v);       // v is strongly connected in this component
            Component[v] = SCC_no;      // Marking SCC nodes to as same component
            if(u == v) break;
        }
}} 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])  continue;              // No Self loop in new graph
            G1[it->second].push_back(Component[v]);
}}} void RunSCC(int V) {
    memset(dfs_num, 0, sizeof(dfs_num)), visited.reset(), dfsCounter = 0, SCC_no = 0;
    for(int i = 1; i <= V; i++) if(dfs_num[i] == 0) tarjanSSC(i);
}
```

```
/* ----------------------------------------- Tree ----------------------------------------- */
/* sTime/in  : starting time of node n
   eTime/out : finishing time of node n
     1
    / \
   5   6
      / \
     7   4
        / \
       2   3
discover_nodes/revIn : {1, 5, 6, 7, 4, 2, 3}
sTime[]/in[]  : {1, 6, 7, 5, 2, 3, 4}              index starts from 1,
eTime[]/out[] : {7, 6, 7, 7, 2, 7, 4}              i'th index contains start time of i'th node
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
range updates can be performed in range of start time and end time of a node */

/* ----------------------------- DFS Timing, Child Finding, LCA----------------------------- */
int cnt = 0;                                                                  // cnt is used for timer
void dfs(int u, int p) {
    in[u] = ++cnt;
    revIn[cnt] = u, par[u][0] = p, lvl[u] = lvl[p]+1;
    for(int i = 1; i <= 20; ++i) par[u][i] = par[par[u][i-1]][i-1];              // used for LCA
    for(auto v : G[u]) { if(v != p) dfs(v, u); }
    out[u] = cnt;
} int LCA(int u, int v) {
    if(lvl[u] < lvl[v]) swap(u, v);
    for(int p = 20; p >= 0; --p) {
        if(lvl[u] - (1 << p) >= lvl[v]) u = par[u][p];
    } if(u == v) return u;
    for(int p = 20; p >= 0; --p) { if(par[u][p] != par[v][p]) u = par[u][p], v = par[v][p]; }
    return par[u][0];
} int dist(int a, int b) {
    return lvl[a] + lvl[b] - 2*LCA(a, b);
}
// LCA if the root changes, [first dfs is done with root 1 or any other fixed node]
int LCA(int u, int v, int root) {                              // root is the new root of the tree
    if(isChild(u, root) and isChild(v, root))   return LCA(u, v);
    if(isChild(u, root) != isChild(v, root))    return root;
    int x = LCA(u, v), y = LCA(u, root), z = LCA(v, root);
    int a = lvl[root] - lvl[x], b = lvl[root] - lvl[y], c = lvl[root] - lvl[z];
    if(a <= b and a <= c) return x;
    if(b <= a and b <= c) return y;
    return z;
}
// Check if one node is child of another node
bool isChild(int child, int par) {                             // returns true if a is child of b
    return ((child == par) or ((in[par] <= in[child]) and (out[par] >= out[child])));
}
// a is upper node (lower level) of path a-b and c is upper node (lower level) of path c-d
```

```
// path a-b and c-d overlaps iff b is a child of c or d or both of c&d
pii overlapPath(int a, int b, int c, int d) {   // returns number of common path of c-d and a-b
    if(not isChild(b, c)) return {0, 0};
    int u = LCA(b, d);                    // u is the lowest point on which c-d and a-b overlaps
    if(level[a]>level[c])   { if(isChild(u, a)) return {a, u}; }
    else                    { if(isChild(u, c)) return {c, u}; }                // c is above a
    return {0, 0};                                                       // no common path found
}
// Finds number of edges if we join nodes a, b and want to find path from c to d
int EdgeCount(int a, int b, int c, int d) {
    int u = LCA(a, b), int v = LCA(c, d);           // connected paths are u->a & u->b
    int ans = distance(c, d, v);                    // query paths are v->c & v->d
    pii tt;                                                             // cases:
    tt = overlapPath(v, c, u, a);                                       // u->a overlaps v->c
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    tt = overlapPath(v, c, u, b);                                       // u->a overlaps v->d
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    tt = overlapPath(v, d, u, a);                                       // u->b overlaps v->c
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    tt = overlapPath(v, d, u, b);                                       // u->b overlaps v->d
    ans -= tt.fi == 0? 0:dist(tt.fi, tt.se, LCA(tt.fi, tt.se));
    return ans;
}
/* ---------------------- Subtree update and query with changed root ----------------------- */
void subTreeUpdate(int u, int root, int val) {              // Subtree update with changed root
    if(u == root)              DS.update(in[1], out[1], val);
    else if(isChild(u, root))  DS.update(in[u], out[u], val);
    else if(isChild(root, u)) {
        int x = getChild(root, u);
        DS.update(in[1], out[1], val), DS.update(in[x], out[x], -val);
    } else DS.update(in[u], out[u], val);
}
// Subtree update with changed root      [the root of update and query doesn't have to be same]
ll getSubTreeSum(int u, int root) {
    if(u == root)           return DS.query(in[1], out[1]);
    if(isChild(u, root))    return DS.query(in[u], out[u]);
    else if(isChild(root, u)) {
        int x = getChild(root, u);
        return DS.query(in[1], out[1]) - DS.query(in[x], out[x]);
    } else return DS.query(in[u], out[u]);
}
/* ----------------------------------- LCA of a subset ----------------------------------- */
// Given a tree, you have to find the LCA of a subset of nodes from the dree
struct LCATree {
    int lca, n, cost;                               // current lca, number of nodes, total edge
    SegTree DS;                                     // DS contains point update and range query
    set<int>nodes;
    void init(int sz) { n = sz, lca = -1, cost = 0; }
    int getPar(int u, int p) {
        for(int i = 20; i >= 0; --i) { if(p & (1 << i)) u = par[u][i]; } // parent sparse table
        return u;
    } int LCA() {
        int u = *nodes.begin(), tot = nodes.size(), v, ret = *nodes.begin();
        int lo = 0, hi = lvl[u]-1;
```

```
        while(lo <= hi) {
            int mid = (lo+hi)>>1, v = getPar(u, mid);
            if(DS.query(1, 1, n, in[v], out[v]) == tot)     hi = mid-1, ret = v;
            else                                            lo = mid+1;
        } return ret;
    } int findChainPar(int u, int t) {                      // finds parent node of u having
        int lo = 0, hi = lvl[u]-1, ret = u, v, mid;         // active child node more than t
        while(lo <= hi) {
            mid = (lo+hi)>>1, v = getPar(u, mid);
            if(DS.query(1, 1, n, in[v], out[v]) > t)    hi = mid-1, ret = v;
            else                                        lo = mid+1;
        } return ret;
    } void addNode(int u) {
        int pstLca = lca;
        nodes.insert(u), DS.update(1, 1, n, in[u], 1);
        if(lca == -1) { lca = u; return; }
        else lca = LCA();
        if(pstLca == lca and query(1, 1, n, in[u], out[u]) == 1) {
            int v = findChainPar(u, 1);   // new LCA is same but the node is on different chain
            cost += lvl[u] - lvl[v];
        } else if(lca != pstLca)  // new LCA changes, newLCA will always be upper from past LCA
            cost += lvl[u] + lvl[pstLca] - 2*lvl[lca]; // also the node u is on different chain
    } void removeNode(int u) {
        int pstLca = lca;
        nodes.erase(u), DS.update(1, 1, n, in[u], -1);
        if(nodes.empty())   { lca = -1, cost = 0; return; }
        else                lca = LCA();
        if(pstLca == lca and query(1, 1, n, in[u], out[u]) == 0) {
            int v = findChainPar(u, 0);
            cost -= lvl[u] - lvl[v];
        } else if(lca != pstLca)
            cost -= lvl[lca] + lvl[u] - 2*lvl[pstLca];
}};
/* Heavy Light Decomopse
   Tree path update/query, there are total log(n) linear chains of a tree */
int parent[MAX], level[MAX], nextNode[MAX], chain[MAX], num[MAX], val[MAX], numToNode[MAX];
int top[MAX], ChainSize[MAX], mx[MAX], ChainNo = 1, all = 1, n;
void dfs(int u, int Parent) {
    parent[u] = Parent, ChainSize[u] = 1;
    for(auto v : G[u]) {
        if(v == Parent)  continue;
        level[v] = level[u]+1;
        dfs(v, u);
        ChainSize[u] += ChainSize[v];
        if(nextNode[u] == -1 || ChainSize[v] > ChainSize[nextNode[u]]) nextNode[u] = v;
}} void hld(int u, int Parent) {
    chain[u] = ChainNo, num[u] = all++; // Giving each nodes a chain number and numbering nodes
    if(ChainSize[ChainNo] == 0) top[ChainNo] = u;                        // first node of chain
    ChainSize[ChainNo]++;
    if(nextNode[u] != -1) hld(nextNode[u], u);                       // Next max chain node exists
    for(auto v : G[u]) {
        if(v == Parent || v == nextNode[u])  continue;
        ++ChainNo; 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]]]) swap(u, v);    // u is the deeper chain
        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;          // build DS using the num[node]
} int main() {                                          // Driver function of HLD
    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);                                      // initialize DS
}
/* ------------------------------------- DSU on TREE ------------------------------------- */
int sz[maxn];
void getsz(int v, int p){
    sz[v] = 1;                                          // every vertex has itself in its subtree
    for(auto u : G[v]) if(u != p) { getsz(u, v); sz[v] += sz[u]; }
}                                                       // add size of child u to its parent(v)
// Heavy-Light-Decomposition Style: nlogn
int cnt[maxn];
bool big[maxn];
void add(int v, int p, int x) {                         // The operation function
    cnt[col[v]] += x;                                   // Perform required operation here
    for(auto u: G[v])
        if(u != p and not big[u])
            add(u, v, x);
} void dfs(int v, int p, bool keep) {           // node, parent, keep the node after dfs execution
    int mx = -1, bigChild = -1;
    for(auto u : G[v])                                      // finding big child with maximum nodes
       if(u != p && sz[u] > mx)      mx = sz[u], bigChild = u;
    for(auto u : G[v])                              //  dfs on small childs and clear them from cnt
        if(u != p && u != bigChild)  dfs(u, v, 0);
    if(bigChild != -1)
        dfs(bigChild, v, 1), big[bigChild] = 1;     // bigChild marked and not cleared from cnt
    add(v, p, 1);
// Answer execution : cnt[c] is the number of vertices in subtree of vertex v that has color c.
    if(bigChild != -1)  big[bigChild] = 0;
    if(keep == 0)       add(v, p, -1);
}
// Map Style: n(logn)^2
map<int, int> *cnt[maxn];
void dfs(int v, int p){
    int mx = -1, bigChild = -1;
    for(auto u : G[v]) {
       if(u != p) {
          dfs(u, v);
```

```
           if(sz[u] > mx) mx = sz[u], bigChild = u;
    }}
    if(bigChild != -1)  cnt[v] = cnt[bigChild];                   // Copies pointer of bigchild
    else                cnt[v] = new map<int, int> ();        // Create empty pointer container
    (*cnt[v])[ col[v] ] ++;
    for(auto u : G[v])
       if(u != p && u != bigChild) {
           for(auto x : *cnt[u]) (*cnt[v])[x.first] += x.second;
       } // (*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c.
}
// Vector Style: nlogn
vector<int> *vec[maxn];
int cnt[maxn];
void dfs(int v, int p, bool keep){
    int mx = -1, bigChild = -1;
    for(auto u : G[v]) { if(u != p && sz[u] > mx) mx = sz[u], bigChild = u; }  // bigChild mark
    for(auto u : G[v]) { if(u != p && u != bigChild)  dfs(u, v, 0); }       // traverse non big
    if(bigChild != -1) { dfs(bigChild, v, 1), vec[v] = vec[bigChild]; }        // pointer copy
    else               { vec[v] = new vector<int> (); }
    vec[v]->push_back(v), cnt[ col[v] ]++;
    for(auto u : G[v])
       if(u != p && u != bigChild)
           for(auto x : *vec[u]) { cnt[ col[x] ]++, vec[v] -> push_back(x); }
    // (*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c.
    if(keep == 0) { for(auto u : *vec[v]) cnt[col[u]]--; }
}

/* --------------------------------------- String --------------------------------------- */
/* ------------------------------------ Aho-Corasick ------------------------------------
Complexity : O(n+m+z)
Finds multiple patterns in a given string with positions and number of occrances of each
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 total 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 build(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 chars.
        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, c = nextInput - lowestChar;
    while(g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}
int cnt[TOTKEY];
void Matcher(const 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;
int main() {
    int n;
    cin >> n >> text;               // n: number of patterns, text: the main string
    while(n--) {
        cin >> str;                 // str: the patterns which are to be found in 'text'
        keywords.push_back(str);
    } build(keywords);
    Matcher(keywords, text);
    cout << "Matches " << Case << ":\n";
    for(int i = 0; i < (int)keywords.size(); ++i)
        cout << cnt[i] << "\n";
    return 0; }
```

```
/* ------------------------------------  Suffix Array -----------------------------------
Complexity : N log^2(N)
Sorts all suffixes in lexicographical order, finds their Longest Common Prefix using Kasai.
Approaches:
1. Number of unique substrings: Sum of lengths of all suffixes - Sum of all LCP,
   Check totalUniqueSubstr() function
2. Minimum lexicographical rotation: Perform Kasai on input string 'S' as 'SS', the minimum
   Suffix rank from index 0-|S| is the answer. rotation -> abcd -> bcda -> cdab -> dabc
3. LCP of two index i, j of string S is the minimum of subarray LCP[rank[i], ..., rank[j]]
4. Lontest Common Substring of multiple string: let S1, S2, S3 be strings. Build new string,
   S = S1+#+S2+$+S3. Perform a sliding window on the LCP array from lower to higher rank, such
   that the window contains suffixes of the three strings. Answer will be the minimum LCP of
   the sliding window.
*/
struct suffix {
    int idx;
    pii rank;
    bool operator < (suffix x) {
        return rank < x.rank;
}};
int order(char x) {
    if(isdigit(x)) return x - '0';
    else if(isupper(x)) return x-'A'+10;
    else if(islower(x)) return x-'a'+36;
    else return 110;
}
int idxToRank[MAX];                          // Index to suffix rank/lexicographicalindex mapping
suffix suff[MAX];                            // Rank is the legicographical index of each suffix
// Adding a '~' after the string takes the longer lenth higher of the SA
void SuffixArray(int len, char str[]) {
    for(int i = 0, j = 1; i < len; ++i, ++j) {
        suff[i].idx = i, idxToRank[i] = 0;              // Initialize value of index i, and i+1

        suff[i].rank.fi = order(str[i]),    suff[i].rank.se = (j<len) ? order(str[j]):-1;
    } sort(suff, suff+len);                             // Out of range position assigned as -1
    for(ll k = 4; k < (2*len); k *= 2) {             // Assigning new first rank for all suffix
        int rank = 0, prevRank = suff[0].rank.fi;         // K is the size of each suffix block
        suff[0].rank.fi = 0, idxToRank[suff[0].idx] = 0;
        for(int i = 1; i < len; ++i) {
            if(suff[i].rank == make_pair(prevRank, suff[i-1].rank.se)) {
                prevRank = suff[i].rank.fi;
                suff[i].rank.fi = rank;
            } else {
                prevRank = suff[i].rank.fi;
                suff[i].rank.fi = ++rank;
            } idxToRank[suff[i].idx] = i;
        } for(int i = 0; i < len; ++i) {
            int nxtIdx = suff[i].idx + k/2;
            suff[i].rank.se = (nxtIdx < len) ? suff[idxToRank[nxtIdx]].rank.fi:-1;
        } sort(suff, suff+len);
    } for(int i = 0; i < len; ++i)
        idxToRank[suff[i].idx] = i;
}
// Optimized Suffix Array
```

```
// Complexity : N log(N)
int o[2][MAX], t[2][MAX];
int idxToRank[MAX], rankToIdx[MAX], A[MAX], B[MAX], C[MAX], D[MAX];
void SuffixArray(char str[], int len, int maxAscii = 256) {
    int x = 0;
    memset(A, 0, sizeof A), memset(C, 0, sizeof C), memset(D, 0, sizeof D);
    memset(o, 0, sizeof o), memset(t, 0, sizeof t);
    for(int i = 0; i < len; ++i) A[(str[i]-'a')] = 1;
    for(int i = 1; i < maxAscii; ++i) A[i] += A[i-1];
    for(int i = 0; i < len; ++i) o[0][i] = A[(int)(str[i]-'a')];
    for (int j = 0, jj = 1, k = 0; jj < len && k < len; ++j, jj <<= 1) {
        memset(A, 0, sizeof A), memset(B, 0, sizeof B);
        for(int i = 0; i < len; ++i) {
            ++A[ t[0][i] = o[x][i] ];
            ++B[ t[1][i] = (i+jj<len) ? o[x][i+jj] : 0 ];
        } for(int i = 1; i <= len; ++i) {
            A[i] += A[i-1];
            B[i] += B[i-1];
        }
        for(int i = len-1; i >= 0; --i)
            C[--B[t[1][i]]] = i;
        for(int i = len-1; i >= 0; --i)
            D[--A[t[0][C[i]]]] = C[i];
        x ^= 1, o[x][D[0]] = k = 1;
        for(int i = 1; i < len; ++i)
            o[x][D[i]]= (k += (t[0][D[i]] != t[0][D[i-1]] || t[1][D[i]] != t[1][D[i-1]]));
    } for(int i = 0; i < len; i++) {
        idxToRank[i] = o[x][i]-1;
        rankToIdx[o[x][i]-1] = i;
}}
// Longest Common Prefix: Kasai's Algorithm
// Complexity: O(n)
int lcp[MAX];                          // LCP[i] contains LCP of index i and i-1
void Kasai(char str[], int len) {      // Matches Same charechters with i'th rank & (i+1)'th rank
    int match = 0;
    for(int idx = 0; idx < len; ++idx) {
        if(idxToRank[idx] == len-1) {
            match = 0;
            continue;
        } int nxtRankIdx = rankToIdx[idxToRank[idx]+1];
        int p = idx+match, q = nxtRankIdx+match;
        while(p < len and q < len and str[p] == str[q])
            ++p, ++q, ++match;
        lcp[nxtRankIdx] = match;               // the lcp match of i'th & (i+1)'th is stored in
        match -= (match > 0);                  // the index of (i+1)'th suffix's index
}}
int consecutiveMaxLCP(int idx, int len) {    // Finds max LCP of index idx and the total string
    int r = idxToRank[idx], ret = lcp[idx];                 //   comparing with the next rank
    if(r+1 < len) ret = max(ret, lcp[suff[r+1].idx]);       //         string of idx's string
    return ret;
}
int totalUniqueSubstr(int len) {                     // Returns total number of unique substring
    int ans = 0;
    for(int rank = 0; rank < len; ++rank) {
```

```
        int idx = suff[rank].idx;
        ans += len-idx;
        if(rank != 0) ans -= lcp[idx];
    } return ans;
}
// Longest Common Prefix [Sparse Table after running Kasai]
int table[MAX][14], lg[MAX];
void buildSparseTableRMQ(int n) {                          //  O(n Log n)
    for(ll i = 0; 1LL << i < n; i++)    lg[1LL << i] = i;
    for(ll i = 1; i < n; i++)           lg[i] = max(lg[i], lg[i - 1]);
    for(int i = 0; i < n; ++i)          table[i][0] = i;
    for(int j = 1; (1 << j) <= n; ++j) {                    // j is the power : 2^j
        for(int i = 0; i + (1 << j) - 1< n; ++i) {
            if(lcp[rankToIdx[table[i][j-1]]] < lcp[rankToIdx[table[i + (1 << (j-1))][j-1]]])
                table[i][j] = table[i][j-1];
            else
                table[i][j] = table[i + (1 << (j-1))][j-1];
}}}
int sparseQueryRMQ(int l, int r) {          // Gives LCP of index l, r in O(1)
    l = idxToRank[l], r = idxToRank[r];     // Remove this line if rankUp or rankDown is used
    if(l > r) swap(l, r);
    ++l;
    int k = lg[r - l + 1];                  // log_2 segment;
    return min(lcp[rankToIdx[table[l][k]]], lcp[rankToIdx[table[r - (1 << k) + 1][k]]]);
}
// Gives Upper (lower rank) for which the Range minimum LCP is tlen
// Call : 0, PosRank, strlen, totstring_len
int rankUP(int lo, int hi, int tlen, int len) {
    int mid, ret = hi, pos = hi;
    --hi;
    while(lo <= hi) {
        mid = (lo+hi)>>1;
        if(sparseQueryRMQ(mid, pos) >= tlen)    hi = mid-1, ret = mid;
        else                                    lo = mid+1;
    } return ret;
}
// Gives Lower (higher rank) for which the Range minimum LCP is tlen
// Call : PosRank, len-1 strlen, totstring_len
int rankDown(int lo, int hi, int tlen, int len) {
    int mid, ret = lo, pos = lo;
    ++lo;
    while(lo <= hi) {
        mid = (lo+hi)>>1;
        if(sparseQueryRMQ(mid, pos) >= tlen)    lo = mid+1, ret = mid;
        else                                    hi = mid-1;
    } return ret;
}
/* ----------------------------------------- Hashing -----------------------------------------
Eqn : s[i] * p^i + s[i+1] * p^(i+1) ...
Hash powers starting from x and y, matched by multiplying with Power[MAX-x] and Power[MAX-y]
*/
const ll p = 31;
const ll mod1 = 1e9+9, mod2 = 1e7+7;
// --------------- DOUBLE HASH GENERATORS ---------------
```

```
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-l].se)%mod2;
    return H;
}
// Dynamic Hash supports replacing and deletion of charachter
struct DynamicHash {
    struct HashTree {                               // Data Structure of dynamix hash
        vector<ll>sum, propSum, propMul;
        ll mod, len;
        inline ll add(ll a, ll b) { return (a+b)%mod; }
        inline ll mul(ll a, ll b) { return (a*b)%mod; }
        void resize(int n, ll _mod, ll arr[]) {
            sum.resize(4*n), propSum.resize(4*n);
            propMul.resize(4*n), mod = _mod, len = n;
        } void pushDown(int child, int par) {                      // just push down the values
            propSum[child] = mul(propSum[child], propMul[par]);
            propSum[child] = add(propSum[child], propSum[par]);
            propMul[child] = mul(propMul[child], propMul[par]);
        } void init(int pos, int l, int r, ll arr[]) {                 // Call resize first!!!
            sum[pos] = propSum[pos] = 0, propMul[pos] = 1;
            if(l == r) { sum[pos] = arr[l]; return; }
            int mid = (l+r)>>1;
            init(pos<<1, l, mid, arr), init(pos<<1|1, mid+1, r, arr);
            sum[pos] = add(sum[pos<<1], sum[pos<<1|1]);
        } void propagate(int pos, int l, int r) {             // sets and pushes values to child
            if(propMul[pos] == 1 and propSum[pos] == 0) return;
            sum[pos] = add(mul(sum[pos], propMul[pos]), mul(r-l+1, propSum[pos]));
            if(l == r) { propMul[pos] = 1, propSum[pos] = 0; return; }
            pushDown(pos<<1, pos), pushDown(pos<<1|1, pos);
            propMul[pos] = 1, propSum[pos] = 0;
        } void update(int pos, int l, int r, int L, int R, ll val, int type) {
            propagate(pos, l, r);
            if(r < L or R < l) return;
            if(L <= l and r <= R) {
                if(type == 0)                              // add val in [L, R]
                    propSum[pos] = add(propSum[pos], val);
```

```
                else if(type == 1) {                        // multiply val in [L, R]
                    propSum[pos] = mul(propSum[pos], val);
                    propMul[pos] = mul(propMul[pos], val);
                } else if(type == 2)                          // set all value = val
                    propSum[pos] = val, propMul[pos] = 0;
                propagate(pos, l, r);
                return;
            } int mid = (l+r)>>1;
            update(pos<<1, l, mid, L, R, val, type), update(pos<<1|1, mid+1, r, L, R, val, type);
            sum[pos] = add(sum[pos<<1], sum[pos<<1|1]);
        } ll query(int pos, int l, int r, int L, int R) {
            propagate(pos, l, r);
            if(r < L || R < l) return 0;
            if(L <= l && r <= R) return sum[pos];
            int mid = (l+r)>>1;
            return add(query(pos<<1, l, mid, L, R), query(pos<<1|1, mid+1, r, L, R));
        }
        ll query(int l, int r)              { return query(1, 1, len, l, r); }
        void add(int l, int r, ll val)      { update(1, 1, len, l, r, val, 0); }
        void mul(int l, int r, ll val)      { update(1, 1, len, l, r, val, 1); }
        void set(int l, int r, ll val)      { update(1, 1, len, l, r, val, 2); }
    };
    pair<HashTree, HashTree>H;
    ordered_set<int> indexGen;
    const ll p1 = 31, modInvP1 = 838709685;
    const ll p2 = 51, modInvP2 = 1372550;
    const ll mod1 = 1e9+9, mod2 = 1e7+7;
    ll LIM, len;
    vll Power;
    void init(string &str) {
        LIM = str.size() + 100;
        PowerGen(LIM+100);
        ll h1 = 0, h2 = 0;
        len = SIZE(str);
        indexGen.clear();
        H.first.init(len+5, mod1), H.second.init(len+5, mod2);
        indexGen.insert(0);
        for(int i = 1; i < len; ++i) {                  // assuming string starts from index 1
            h1 = ((str[i] - 'a' + 1) * Power[i].first)%mod1;
            h2 = ((str[i] - 'a' + 1) * Power[i].second)%mod2;
            H.first.add(i, i, h1),  H.second.add(i, i, h2);
            indexGen.insert(i);
    }}
    int GetPos(int idx) { return *indexGen.at(idx); }
    void PlaceChar(int idx, char newChar) {             // Place/Replace charachter at idx
        int StrIdx = GetPos(idx);
        ll newVal1 = ((newChar-'a'+1)*Power[idx].first)%mod1;
        ll newVal2 = ((newChar-'a'+1)*Power[idx].second)%mod2;
        H.first.set(StrIdx, StrIdx, newVal1),   H.second.set(StrIdx, StrIdx, newVal2);
        str[StrIdx] = newChar;
    } void RemoveChar(int pos) {                          // Remove charachter at pos
        int idx = GetPos(pos);
        H.first.set(idx, idx, 0),           H.second.set(idx, idx, 0);
        H.first.mul(idx+1, len, modInvP1),  H.second.mul(idx+1, len, modInvP2);
```

```
        indexGen.erase(indexGen.at(pos));
    } 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 * p1)%mod1;
            Power[i].second = (Power[i-1].second * p2)%mod2;
    }} ll SubStrHash(int l, int strLen, bool first = 1) {
        int LL = GetPos(l), RR = GetPos(l+strLen-1);
        ll hash = first ? H.first.query(LL, RR):H.second.query(LL, RR);
        return (hash * (first?Power[LIM-l].first:Power[LIM-l].second))%(first?mod1:mod2);
    } ll GetHash(int l, int r) {
        return H.first.query(GetPos(l), GetPos(r));
}};
/* ---------------------------------------- 2D Hash ---------------------------------------- */
// For row, column (i,j) prime power is somthing like p^(ij)
const int lineOffset = 1010;                        // use the 2DLim to distinguish between rows
vector<vll> Gen2DHash(int r, int c, char str[][1010]) {                    // row, column, string
    vector<vll> hash(r);
    for(int i = 0, offset = 0; i < r; ++i, offset += lineOffset) {
        pll h = {0, 0};                             // Powers of every row r starts from r*offset
        for(int j = 0; j < c; ++j) {
            h.first = ((str[i][j] - 'a' + 1)*Power[j+offset].first)%mod1;
            h.second = ((str[i][j] - 'a' + 1)*Power[j+offset].second)%mod2;
            hash[i].push_back(h);
    }} for(int i = 0; i < r; ++i) {
         for(int j = 0; j < c; ++j) {
            if(i > 0) {
                hash[i][j].first = (hash[i][j].first + hash[i-1][j].first)%mod1;
                hash[i][j].second = (hash[i][j].second + hash[i-1][j].second)%mod2;
            } if(j > 0) {
                hash[i][j].first = (hash[i][j].first + hash[i][j-1].first)%mod1;
                hash[i][j].second = (hash[i][j].second + hash[i][j-1].second)%mod2;
            } if(i > 0 and j > 0) {
                hash[i][j].first = (hash[i][j].first - hash[i-1][j-1].first + mod1)%mod1;
                hash[i][j].second = (hash[i][j].second - hash[i-1][j-1].second + mod2)%mod2; }
            hash[i][j].first = (hash[i][j].first)%mod1;
            hash[i][j].second = (hash[i][j].second)%mod2;
    }} return hash;
}
const ll LIM = 1025000;
pll SubHash2D(vector<vll> &H, int x, int y, int r, int c) {           // Generates hash which's
    int xx = x+r-1, yy = y+c-1;                                       // upper-left point = (x, y)
    pll ret = H[xx][yy];                                  // lower right point = (x+r-1, y-c-1)
    if(x > 0) {
        ret.first = (ret.first - H[x-1][yy].first + mod1)%mod1;
        ret.second = (ret.second - H[x-1][yy].second + mod2)%mod2;
    } if(y > 0) {
        ret.first = (ret.first - H[xx][y-1].first + mod1)%mod1;
        ret.second = (ret.second - H[xx][y-1].second + mod2)%mod2;
    } if(x > 0 and y > 0)
        ret.first += H[x-1][y-1].first, ret.second += H[x-1][y-1].second;
    ret.first = ret.first%mod1, ret.second = ret.second%mod2;
    ret.first = (ret.first*Power[LIM-(x*lineOffset+y)].first)%mod1;
```

```
    ret.second = (ret.second*Power[LIM-(x*lineOffset+y)].second)%mod2;
    return ret;
}
/* ---------------------------- Knuth Morris Pratt - KMP ----------------------------
 Complexity : O(String + Token)
 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])
            table[i++] = ++len;
        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
    while (i < strLen) {                                             // j : pattern index
        if(str[i] == pat[j]) i++, j++;
        if(j == patLen) {
            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
}}}
/* p is the pattern where table[] is the previously made prefix-table of pattern
   For any index idx the nxt[idx][j] returns the new index idx where the index
   should point next, this optimizes the kmp in linear time */
void getState(string &p, int table[], int nxt[][27]) {
    for(int i = 0; i < p.size(); ++i) {
        for(int j = 0; j < 26; ++j) {
            if(p[i]-'a' == j)   nxt[i][j] = i+1;
            else                nxt[i][j] = i == 0 ? 0 : nxt[table[i-1]][j];
}}}
/* check function using nxt[idx][j]
 idx is the index from which the string should start matching with the pattern
 by default idx = 0, also it refers the last index of the pattern to which
 the string matched */
int match(string &s, int table[], int nxt[][27], int &idx) {
    int ans = 0;
    for(char c : s) {
        idx = nxt[idx][c-'a'];
        if(idx == p.size())
            ++ans, idx = table[idx-1];
    }
return ans; }

/* --------------------------------------------- Math -------------------------------------
   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 sieveGen(ll N) {                                   // Faster, will generate all primes <= N
    isPrime.set();
    isPrime[0] = isPrime[1] = 0;
    for(ll i = 3; i*i <= N; i+=2) {
        if(isPrime[i]) { for(ll j = i*i; j <= N; j += i) isPrime[j] = 0; }
    } primes.push_back(2);
    for(int i = 3; i <= N; i+=2) { if(isPrime[i]) primes.push_back(i); }
}
// Sublinear Prime Factorization
int pd[MAX];                      // Contains minimum prime factor/divisor, for primes pd[x] = x
vector<int>primes;                // Contains prime numbers
void sublinearSieve(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 < primes.size() && primes[j] <= pd[i] && i*primes[j] <= N; ++j)
            pd[i*primes[j]] = primes[j];
}}
// Basic prime factor
vll primeFactor(ull n) {
    vll 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];
    }} if(n != 1) factor.push_back({n, 1});
    return factor;
} vi fastFactorize(int x) {
    vector<int>factor;
    while(x > 1) { if(pd[x] != 0) {
        factor.push_back(pd[x]);
        x /= pd[x];
    }} return factor;
} vll factorialPrimeFactor(int n) {                     // prime factorization of factorials (n!)
    vll V;
    for(int i = 0; i < primes.size() && primes[i] <= n; i++) {
        ll 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;
}
// if n = p1^a1 * p2^a2,... then NOD = (a1+1)*(a2+1)*...
int NumberOfDivisors(long long n) {
    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;
}
/* 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 millerTest(int d, int n) {
    int a = 2 + rand() % (n - 4);           // Pick a random number in [2..n-2].
    int x = powMod(a, d, n);                // Compute a^d % n
    if (x == 1  || x == n-1)    return 1;
    while (d != n-1) {
        x = (x * x) % n, d *= 2;
        if (x == 1)      return 0;
        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++) { if(miillerTest(d, n) == 0) return 0; }     // Iterate k times
    return 1;
}
ll powerMOD(ll x, ll y, ll MOD) {                       // Can find modular inverse by a^(MOD-2),
    ll res = 1;                                         //  a and MOD must be co-prime
    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;
}
// calculate A mod B, where A : 0<A<(10^100000) (or greater)
ll afterMod(string str, ll mod) {
    ll ans = 0;
    for(auto it = str.begin(); it != str.end(); it++)                       // mod from MSM to LSB
        ans = (ans*10 + (*it -'0')) % mod;
return ans; }
// Exponent of Big numbers (N^P % M)  [where N and P is bigger strings and M is 64 bit integer]
ll bigExpo(char *N, char *P, ll M) {
    ll 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 Euclid
   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, and y_ans = y - (a/d)n;  where n is an integer
   Solution only exists if d | c (i.e : c is divisible by d) */
```

```
ll gcdExtended(ll a, ll b, ll *x, ll *y) {
    if (a == 0) {  *x = 0, *y = 1; return b; }
    ll x1, y1, gcd = gcdExtended(b%a, a, &x1, &y1);
    *x = y1 - (b/a) * x1, *y = x1;euc
    return gcd;
} ll modInverse(ll a, ll mod) {
    ll x, y, g = gcdExtended(a, mod, &x, &y);
    if(g != 1) return -1;                   // Moduler Inverse doesnt exist!
    return (x%mod + mod) % mod;
}
/* ------------------------------------ Math Formulas ------------------------------------ */
// Counts number of values in range [l, r] for which dividing by x returns mod value modVal
ll GetModVals(ll l, ll r, ll modVal, ll x) {
    ll hi = floor((r-modVal)/(double)x), low = ceil((l-modVal)/(double)x);
    return hi-low+1;
}
// Find the number of b for which [b1, b2] | [a1, a2]
int FindDivisorInRange(int a1, int a2, int b1, int b2) {
    return (__gcd(abs(a1 - a2), abs(b1 - b2)) + 1);
}
// 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), common_divisor = n / lcm - (m - 1) / lcm;
    int a_divisor = n / a - (m - 1) / a, b_divisor = n / b - (m - 1) / b;
    return a_divisor + b_divisor - common_divisor;
}
 // Cumulative Sum of Divisors in sqrt(n)
ll cumulativeSumOfDiv(ll 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;
}
// Returns how many times a value P is present in n factorial (n!)
int FactorialCount(int n, int p = 5) {        // Returns number of trailing zero of n! if p = 5
    int ret = 0, r = p;
    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(cnt_2, cnt_5)
    while(n%5 == 0 && n%2 == 0) n /= 5, n /= 2, ++cnt;
    return cnt*p;
}
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
        x = 10LL * x + y;                           // Now calculate the new x and g(x)
        if(y == 0LL) gx++;
    } return fx+1;
}
```

```
/* 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., GCD(i, x) = 1. Phi(4): GCD(1,4)=1, GCD(3,4)=1. so, Phi(4)=2 */
int phi(int n) {                                                               // Computes phi of n
    int result = n;
    for(int p=2; p*p<=n; ++p) {
        if(n % p == 0) {
            while (n % p == 0) n /= p;        // Eleminate all prime factors and their multiple
            result -= result / p;
    }}
    if(n > 1) result -= result / n;                         // If n > 1, then it is also a prime
    return result;
}
int phi[MAX];
void precalPhi(int n) {                                               // Precalculated Euler Totient
    for(int i = 1; i <= n; i++) phi[i] = i;
    for(int p = 2; p <= n; p++) {
        if(phi[p] == p) {
            phi[p] = p-1;
            for(int i = 2*p; i <= n; i += p) { phi[i] = (phi[i]/p) * (p-1); }
}}}
// Combination : Complexity O(k)
ll C(int n, int k) {
    ll c = 1;
    if(k > n - k) k = n-k;              // As n_C_k = n_C_(n-k)
    for(int i = 0; i < k; i++) { c *= (n-i); c /= (i+1); }   // take 1 from n-i in c*(n-i) ways

    return c;               // due to combination rule, we devide with the number of taken value
}                                                         // otherwise it will remain as permutation
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;
}
// 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 <= MAX; 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];
} ll Catalan(int n) {                                              // Catalan(n) = C(2*n, n)/(n+1)
    return C(2*n, n)/(n+1);
}
// Birthday Paradox : returns Number of people required so that probability is >= target
```

```
int BirthdayParadox(int days, int targetPercent = 50) {
    int people = 0;
    double percent = targetPercent/100.0, gotPercent = 1;
    for( ; gotPercent > percent; ++people)  gotPercent *= (days-people-1)/(double)days;
    return people;                    // Formula : 1 - (365/365) * (364/365) * (363/365) * .....
}
/* 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
    return C(n+k-1, k-1);                                          // is no lower limit
}
/* 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 theorem, using Principle of Inclusion Exclution, this answer
 can be found as, Ans = 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);
        } return ret;
}
/* Multinomial : nC(k1,k2,k3,..km) is such that k1+k2+k3+....km = n and ki == kj and ki != kj
 both are 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
 It tells how many ways we can place k1, k2, k3 people in 3 unique teams such that k1+k2+k3=n
 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(fa[0] == 0) {                                       // if k1=k2=k3, then just push k1 once
        fa[0] = 1;
        for(int i = 1; i < MAX; ++i)  fa[i] = (fa[i-1]*i) % MOD;
    } ll k = 1;
    if((int)K.size() == 1)  k = powerMOD(fa[K[0]], N/K[0]);                 // 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
}
// Number of ways to make N/K teams from N people so that each team contais K people
```

```
ll NumOfWaysToPlace(ll N, ll K) {       // If N = 6, then 1,2|3,4|5,6  and  3,4|1,2|5,6 is same
    vector<ll>v;
    v.push_back(K);                                         // Divide by k!, as 1,2|3,4|5,6 and
    return (Multinomial(N, v)*powerMOD(fa[N/K], MOD-2))%MOD;  // 3,4|1,2|5,6 is considered same
}
// Finds how many ways we can place n numbers where r of them are not in their initial place
// Formula: n! - C(n, 1)*(n-1)! + C(n, 2)*(n-2)! ..... + (-1)^r * C(n,r)*(n-r)!
ull partial_derangement(int n, int r) {
    ull ans = f[n];                                                      // Factorial of n!
    for(int i = 1; i <= r; ++i) {
        if(i & 1) ans = (ans%MOD - (C(r, i) * f[n-i])%MOD)%MOD; // Here C(r, i) is because we
        else      ans = (ans%MOD + (C(r, i) *f[n-i])%MOD)%MOD;  // only have to choose from r
        ans = (ans + MOD)%MOD;                                  // elements, not n elements.
    } return ans%MOD;
}
/* 1. Basic Recurrence:
--------------------
f(n) = x*f(n-1) + y*f(n-2) + z*c
-----------------------------------
| x  y  z |   | f(n-1) |   |  f(n)  |
| 1  0  0 | x | f(n-2) | = | f(n-1) |
| 0  0  1 |   |    c   |   |    c   |
-----------------------------------
    T       x      f     =    ans

  2. Even/Odd Seperate Function:
  -----------------------------
f(n) = if n is even: f(n) = x*f(n-1) -y*f(n-2) + c
               else: f(n) = z*f(n-2)
f(1) = f(2) = 1
Build  :
        |x  y  z|           |0  z  0|          |1|
T_even :|1  0  0|     T_odd :|1  0  0|    f(2) :|1|
        |0  0  1|           |0  0  1|          |c|
T : T_even * T_odd
if n is odd then, f(n) :
    n = n-2
    ans = (T^(n/2)) * f(2)
else if n is odd, f(n):
    n = n-2
    ans = T_odd * (T^(n-1)/2) * f(2)
Why this works:
matrix T contains same number of even and odd function calculations
so from start point (here start point is 2 of f(2)), if there exists same number of
even and odd function calculation then calculating power of T is enough.
else we need to multiply the extra T_even or T_odd with T according to the problem
REF: http://fusharblog.com/solving-linear-recurrence-for-programming-contest/

  3. Cumulative Sum:
  ------------------
To calculate cumulative sum, just add another extra row in base matrix T
and carry the previous sum with new function value as well
Example -> Cumulative sum of:
           f(n) = x*f(n-1) + y*f(n-2) + c
```

```
            where, f(1) = f(2) = 1;
Let, S(n) = is the sum of first n values

|1 x y 1|   |S(n-1)|   | S(n) |
|0 x y 1|   |f(n-1)|   | f(n) |
|0 1 0 0| X |f(n-2)| = |f(n-1)|
|0 0 0 1|   |  c   |   |   c  |
-------------------------------
    T     X  f(n-1)  =   f(n)
*/
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);
} MAT pow(MAT x, ll p, int sz) {                           // Power using loop
    if(p == 1) return x;
    MAT ret;
    for(int i = 0; i < sz; ++i) ret.m[i][i] = 1;            // Diagonal Matrix
    while(p > 0) {
        if(p&1) ret = mul(ret, x, sz, sz, sz);
        p = p >> 1, x = mul(x, x, sz, sz, sz);
} return ret; }

/* ----------------------------------------- Flows -----------------------------------------*/
/* #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
   #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(auto v : G[u]) {
        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;             // If there is one node present on the left
            return 1;                           // side of v (Let it be u') and if it is possible
```

```
        } else if(VertexCover(lft[v])) {     // to match u' with another node (not v ofcourse!)
            lft[v] = u, rht[u] = v;          // then we can match this u with v, and u' is
            return 1;                        // 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;
}
/* ---------------------------- MaxFlow (Directed/Undirected) ------------------------------
   Ford-Fulkerson
   Complexity: O(VE^2) */
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;
} void AddEdge(int u, int v, int w) {
    edge[u].push_back(v), edge[v].push_back(u);
    rG[u][v] += w, rG[v][u] += w;
}
/* ------------------------ Min Cost Max Flow (Directed/Undirected) ------------------------
   Edmonds-Karp relabelling + Dijkstra
   Complexity : O(V*V*flow) */
vi G[MAX];
int cost[MAX][MAX], cap[MAX][MAX], dist[MAX], parent[MAX];
bitset<MAX>vis;
```

```
bool Dikjstra(int src, int sink) {
    queue<int>q;
    memset(dist, INF, sizeof dist);
    vis.reset(), q.push(src), vis[src] = 1, dist[src] = 0;   // dist[u] : contains minimum cost
    while(!q.empty()) {
        int u = q.front();
        q.pop(), vis[u] = 0;               // node u is processed and poped out, so set vis = 0
        for(int i = 0; i < (int)G[u].size(); ++i) {
            int v = G[u][i], w = dist[u] + cost[u][v];
            if(cap[u][v] > 0 and dist[v] > w) {     // if capacity exists and can minimize cost
                dist[v] = w, parent[v] = u;
                if(not vis[v]) { q.push(v), vis[v] = 1; }    // check if node v is not in queue
    }}} return dist[sink] != INF;      // this check is because we might insert same node twice
} int MinCostFlow(int src, int sink, int &max_flow) {          // Returns min cost and max flow
    int flow, min_cost = 0;
    max_flow = 0;
    while(Dikjstra(src, sink)) {                            // Max flow does bfs
        flow = INF;
        for(int v = sink; v != src; v = parent[v]) {
            int u = parent[v], flow = min(flow, cap[u][v]);
        } for(int v = sink; v != src; v = parent[v]) {
            int u = parent[v];
            cap[u][v] -= flow, cap[v][u] += flow, cost[v][u] = -cost[v][u];
        } min_cost += dist[sink]*flow, max_flow += flow;                   // cost of this flow
    } return min_cost;                                   // flow = total_cost * actual_flow
} void AddEdge(int u, int v, int _capacity, int _cost) {            // Assuming undirected graph
    G[u].push_back(v), G[v].push_back(u);
    cost[u][v] = cost[v][u] = _cost;                        // Cost of edge u-v
    cap[u][v] = cap[v][u] = _capacity;                      // Capacity of edfe u-v
}

/* --------------------------------------- Ad-Hoc ----------------------------------------- */
/* -------------------------- Longest Increasing/Decrasing Sequence --------------------------
   Non-Printable Version, Complexity : nLog_n */
int LIS(vi v) {                                             // v is the input array
    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())    LIS.push_back(it);           // For Longest Increasing Sequence
        else                    *pIT = it;                   // use lower_bound
    } return 0;
}
/* -------------------------Printable Version-----------------------
   DP + BinarySearch (nLog_n)    INPUT ARRAY : {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(vi &v, vi &idx) {           // v contains input values and idx contains
    if(v.empty()) return;                // index of the LIS values
    vector<int> dp(v.size());            // The memoization part, remembers what index is the
    idx.push_back(0);                    // previous index if any value is inserted or modified
    int l, r;                            // Carrys index of values
    for(int i = 1; i < (int)v.size(); i++) {
        if(v[idx.back()] <= v[i]) {          // Replace < with <= for non-decreasing subsequence
            dp[i] = idx.back(), idx.push_back(i);
            continue;                              // Binary search is done on idx (not in v)
        } for(l = 0, r = idx.size()-1; l < r; ) { // Binary search to find the smallest element
```

```
            int mid = (l+r)/2;                    // referenced by idx which is just bigger
            if(v[idx[mid]] <= v[i])  l = mid+1;   // than v[i] (UpperBound(v[i]))
            else                     r = mid;     // replace <= with < if non-decreasing needed
        } if(v[i] < v[idx[l]]) {                  // Update idx if new value is smaller then
            if(l > 0) dp[i] = idx[l-1];           // previously referenced value
            idx[l] = i;
    }} for(l = idx.size(), r = idx.back(); l--; r = dp[r])
        idx[l] = r;
}
/* --------------------------------------- 1D Max Sum ---------------------------------------
   Algorithm : Jay Kadane, Complexity : O(n) */
int maxSum1D(int A[], int len) {
    int sum = 0, ans = 0;
    for(int i = 0; i < len; i++) {
        sum += A[i];
        ans = max(sum, ans);                       // Always take the larger sum
        sum = max(sum, 0);                         // if sum is negative, reset it (greedy)
    } return ans;
}
/* --------------------------------------- 2D Max Sum ---------------------------------------
   Algorithm : DP, Inclusion Exclusion, Complexity : O(n^4) */
int maxSum2D(int A[][100], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; 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 < n; i++) {                  // i & j are the start coordinate of sub-rect
        for(int j = 0; j < n; j++) {
            for(int k = i; k < n; k++) {         // k & l are the finish coordinate of sub-rect
                for(int l = j; l < n; 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];          // Inclusion exclusion
                    maxSubRect = max(subRect, maxSubRect);
    }}}} return maxSubRect;
}
/* -------------------------------------- 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, mid2 = high - (high - low) / 3;
        ll cost1 = f(mid1), 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;
}
/* ------------------------------------- Merge Sort -------------------------------------- */
void MergeSort(int 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] = INF;                   // INF value in both array
    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;                // Add in else if Min Number of Swaps needed
}} void Divide(int arr[], int l, int r) {                           // Call as Divide(v, 0, len)
    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);
}

/* ---------------------------------------- Geometry --------------------------------------- */
struct point {                                  // Integer Point
    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 {                                  // Float Point
    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
    double rad = DEG_to_RAD(theta);                     // (theta is in degree)
    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
    return (p1.x*(p2.y-p3.y) + p2.x*(p3.y-p1.y) + p3.x*(p1.y-p2.y));   // points are clockwise,
}                                                                // Negative for Anti-Clockwise
/*  if(slope==0): They are all colinear
    if(slope>0) : They are all clockwise
    if(slope<0) : They are counter clockwise  */
double whichSide(point p, point q, point r) {                   // returns on which side point
    double slope = (p.y-q.y)*(q.x-r.x) - (q.y-r.y)*(p.x-q.x);   // r is w.r.t pq line
    return slope;
}
/* ------------------------------------- Lines ------------------------------------- */
struct line { double a, b, c; };                    // ax + by + c = 0  [comes from y = mx + c]
void pointsToLine(point p1, point p2, line &l) {
    if (fabs(p1.x - p2.x) < EPS) l.a = 1.0, l.b = 0.0, l.c = -p1.x;    // vertical line is fine
    else {                                                                  // default values
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);                  // IMPORTANT: we fix the
        l.b = 1.0, l.c = -(double)(l.a * p1.x) - p1.y;                     // value of b to 1.0
}} 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);
    if(fabs(l1.b) > EPS)    p.y = -(l1.a * p.x + l1.c);     // special case: test for vertical
    else                    p.y = -(l2.a * p.x + l2.c);     // line to avoid division by zero
    return 1;
} line perpendicularLine(line l, point p) {         // returns a perpendicular line on l
    line ret;                                       // which goes throuth 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
    if(ret.b != 0) {                                        // by default
        ret.a /= ret.b, ret.c /= ret.b, ret.b = 1;
    } return ret;
}
/* ------------------------------------ Vectors ------------------------------------- */
struct vec {
    double x, y;
    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 cross(vec a, vec b) {                                  // Cross product of two vectors
    return a.x * b.y - a.y * b.x;
} 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)
        return point(a.x + t*(b.x-a.x), a.y + t*(b.y-a.y));                 // t = [-inf, +inf]
}};
// Returns the distance from p to the line defined by  two points a and b
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 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)));
} bool collinear(point p, point q, point r) {               // returns true if point r is on the
    return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;             // same line as the line pq
}
/* ---------------------------------------- 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
    int dis = dist(point(c1.x, c1.y), point(c2.x, c2.y)); //circle is fully into another circle
    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)
    return acos((AB*AB + BC*BC - CA*CA)/(2*AB*BC));            // opposite of side CA
} double rInCircle(double ab, double bc, double ca) {             // Returns radius of inCircle
    return TriangleArea(ab, bc, ca) / (0.5 * (ab + bc+ ca));      // of a triangle
} 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
    double ab = dist(a, b), bc = dist(b, c), ca = dist(c, a);         // radius of circumcircle
    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;
}
```