GT Anaar

Animesh Fatehpuria, Matthew Fahrbach, Rares Cristian, Majid Farhadi

November 10, 2017

Contents

1 Data Structures **4**
1.1 2D Segment Tree 4
1.1.1 GCD 2D Segment Tree 4
1.2 Fenwick Tree 6
1.2.1 2D BIT 6
1.3 HLD Trick 8
1.3.1 HLD Trick 8
1.4 LIS Tricks 10
1.4.1 ZIPLine LIS 10
1.5 Link Cut Tree 12
1.5.1 Link Cut Tree 12
1.6 Merge Sort Tree 17
1.6.1 Merge Sort Tree 17
1.7 Persistent Segment Trees 19
1.7.1 2D PST 19
1.7.2 Non Pointer PST 21
1.7.3 Pointer PST 23
1.8 SQRT Decomposition 24
1.8.1 Mos on Trees 24
1.8.2 Mos with Updates 27
1.9 Sparse Table 30
1.9.1 Sparse Table 30
1.10 Treaps 30
1.10.1 Implicit Treap Cut And Paste 30
1.10.2 Implicit Treap Reverse Subarray 32
1.10.3 Normal Treap Fancy Queries 35
1.10.4 Normal Treap Less Than K 38
1.10.5 Persistent Implicit Treap 40
1.11 Trie 43
1.11.1 Binary Trie 43
1.11.2 Persistent Trie 45
1.12 Union Of Rectangles 47
1.12.1 Area Of Union Of Rectangles 47
1.13 Wavelet Tree 50
1.13.1 Wavelet Tree 50

2 Flows and 2SAT **53**
2.1 2SAT 53
2.1.1 2SAT Template 53
2.2 Bipartite Matching 55
2.2.1 BPM Template 55
2.2.2 Semi Dynamic Matching 60
2.3 Directed Graphs (Matching) 62

2.3.1 Maximum Antichain Dilworths . . . 62
2.3.2 Min Path Cover General . . . 65
2.4 General Graph Matching . . . 68
2.4.1 Edmonds Matching . . . 68
2.5 Max Flow and Min Cut . . . 71
2.5.1 Dinics . . . 71
2.5.2 Min Cut . . . 74
2.6 Min Cost Max Flow . . . 76
2.6.1 Mcmf Bellman Ford . . . 76
2.6.2 Mcmf SPFA . . . 79
2.6.3 Mcmf Trees . . . 81

3 Game Theory **85**
3.1 A Partitioning Game . . . 85
3.2 A String Game . . . 86
3.3 Prime Game . . . 87

4 Geometry **88**
4.1 3D . . . 88
4.1.1 3D Template . . . 88
4.2 Circles . . . 95
4.2.1 Circles . . . 95
4.2.2 Minimum Enclosing Circle . . . 98
4.3 Fundamentals . . . 98
4.3.1 Template . . . 98
4.4 Lines . . . 101
4.4.1 Lines . . . 101
4.4.2 Segment Intersection . . . 101
4.5 Points . . . 105
4.5.1 Closest Pair . . . 105
4.5.2 Delaunay . . . 106
4.5.3 Diameter . . . 108
4.6 Polygons . . . 109
4.6.1 Convex Cut . . . 109
4.6.2 Convex Hull . . . 110
4.6.3 Polygon Union Area . . . 111
4.6.4 Polygon Union Intersection . . . 112

5 Graphs and Trees **115**
5.1 Auxiliary Tree . . . 115
5.1.1 Auxiliary Tree . . . 115
5.2 Block Cut Tree . . . 117
5.2.1 Block Cut Tree . . . 117
5.3 Bridge Tree . . . 121
5.3.1 Bridge Tree . . . 121
5.4 Centroid Decomposition . . . 122
5.4.1 Centroid Decomposition (Race) . . . 122
5.4.2 Centroid Subtrees . . . 124
5.4.3 Tree Queries Centroid . . . 125
5.4.4 YATP . . . 127
5.5 DAG . . . 131
5.5.1 Dominators (DAG) . . . 131
5.5.2 Semidynamic Longest Path . . . 132
5.6 Directed MST . . . 134
5.6.1 Edmonds . . . 134
5.7 Euler Path . . . 136
5.7.1 Euler Path . . . 136

5.7.2 Make Graph Eulerian 137
5.8 Gomory Hu Tree 139
5.8.1 Gomory Hu Tree 139
5.9 Heavy Light Decomposition 142
5.9.1 QTREE HLD 142
5.9.2 Subtrees and Paths HLD 144
5.10 Reachability Tree 146
5.10.1 Reachability Tree 146
5.11 Shortest Paths 149
5.11.1 Count Shortest Paths 149
5.11.2 Semidynamic Shortest Path Tree 150
5.12 Strongly Connected Components 153
5.12.1 SCC 153

6 Maths and DP **154**
6.1 Bitmask DP 154
6.1.1 TSP 154
6.2 CRT 156
6.2.1 Chinese Remainder Theorem 156
6.3 Combinatorics 157
6.3.1 Combinatorics 157
6.3.2 Equivalence Class DP 158
6.3.3 Gerald and Giant Chess 161
6.3.4 Lucas Theorem 162
6.4 DP Optimizations 163
6.4.1 Divide and Conquer 163
6.4.2 Semidynamic Convex Hull Trick 164
6.5 FFT 167
6.5.1 FFT 167
6.5.2 NTT 169
6.6 Integration 172
6.6.1 Revolving Curve 172
6.6.2 Simpsons Rule 174
6.7 Linear Algebra 174
6.7.1 Basis 174
6.7.2 Gaussian with Rationals 176
6.7.3 Modular Gaussian Elimination 179
6.7.4 RREF 182
6.7.5 XOR Shortest Path 183
6.8 Linear Programming 184
6.8.1 Simplex 184
6.9 Matrix Exponentiation 187
6.9.1 Matrix Exponentiation Advanced 187
6.9.2 Matrix Exponentiation 189
6.10 Miscellaneous Algebra 191
6.10.1 Algebra 191
6.10.2 Lagrange Multipliers 194
6.11 Number Theory 196
6.11.1 Divisor Function 196
6.11.2 Euler Totient 196
6.11.3 Extended Euclid 198
6.11.4 Miller Rabin 198
6.12 Yates DP 199
6.12.1 Compatible Numbers 199
6.12.2 Jzzhu and Numbers 200
6.12.3 Special Pairs 201

7 NP Complete **202**
7.1 Graph Coloring . . . 202
7.2 Max Clique . . . 204

8 Strings **206**
8.1 Aho Corasick . . . 206
8.1.1 Aho Corasick . . . 206
8.2 Hashing . . . 209
8.2.1 2D Hashing . . . 209
8.2.2 String Hashing . . . 212
8.3 KMP . . . 214
8.3.1 KMP . . . 214
8.3.2 LCS KMP DP . . . 214
8.3.3 Optimization KMP DP . . . 216
8.4 Suffix Arrays . . . 218
8.4.1 Distinct Substrings . . . 218
8.4.2 Lexico Smallest Rotation . . . 219
8.5 Z Function . . . 221
8.5.1 Substring Frequency ZFunction . . . 221
8.5.2 Template ZFunction . . . 221

9 Tool Box **223**
9.1 BigIntegers . . . 223
9.2 Date . . . 225
9.3 Math Utilities . . . 226
9.4 String Parsing . . . 227

1 Data Structures

1.1 2D Segment Tree

1.1.1 GCD 2D Segment Tree

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

const int MAX = 275000;
int r, c, t;
vector < int > all_x, all_y;
int type[MAX], a1[MAX], b1[MAX], a2[MAX], b2[MAX];
long long val[MAX];

inline long long gcd(long long X, long long Y) {
    long long tmp;
    while (X != Y and Y != 0) {
        tmp = X;
        X = Y;
        Y = tmp % Y;
    }
    return X;
}

struct node{
    node *left, *right, *outer;
    long long val;
    node(){
        left = right = outer = NULL;
        val = 0LL;
    }
    inline void create(node* &x){
        if(!x) x = new node();
    }
    inline node* update_y(node* &left_x, node* &right_x, int l, int r, int y, long long v, bool isLeaf){
        if(l == r){
            if(isLeaf) val = v;
            else val = gcd((left_x) ? (left_x -> val) : (0), (right_x) ? (right_x -> val) : (0));
            return this;
        }
        int mid = (l + r) >> 1;
        if(mid >= y){
            create(left);
            left = left -> update_y((left_x) ? (left_x -> left) : (left_x),
                    (right_x) ? (right_x -> left) : (right_x), l, mid, y, v, isLeaf);
        }
        else{
            create(right);
            right = right -> update_y((left_x) ? (left_x -> right) : (left_x),
                    (right_x) ? (right_x -> right) : (right_x), mid + 1, r, y, v, isLeaf);
        }
        val = gcd((left) ? (left -> val) : (0), (right) ? (right -> val) : (0));
        return this;
    }
    inline node* update_x(int l, int r, int x, int y, long long v){
        create(outer), create(left), create(right);
        create(left -> outer), create(right -> outer);
        if(l == r){
            outer = outer -> update_y(left -> outer, right -> outer, 1, c, y, v, 1);
            return this;
        }
        int mid = (l + r) >> 1;
        if(mid >= x) left = left -> update_x(l, mid, x, y, v);
        else right = right -> update_x(mid + 1, r, x, y, v);
        outer = outer -> update_y(left -> outer, right -> outer, 1, c, y, v, 0);
        return this;
    }
    inline long long query_y(int l, int r, int b1, int b2){
        if(l > b2 or r < b1) return 0;
        if(l >= b1 and r <= b2) return val;
        int mid = (l + r) >> 1;
        return gcd((left) ? (left -> query_y(l, mid, b1, b2)) : (0),
                (right) ? (right -> query_y(mid + 1, r, b1, b2)) : (0));
    }
    inline long long query_x(int l, int r, int a1, int b1, int a2, int b2){
        if(l > a2 or r < a1) return 0;
        if(l >= a1 and r <= a2) return (outer) ? (outer -> query_y(1, c, b1, b2)) : (0);
        int mid = (l + r) >> 1;
        return gcd((left) ? (left -> query_x(l, mid, a1, b1, a2, b2)) : (0),
                (right) ? (right -> query_x(mid + 1, r, a1, b1, a2, b2)) : (0));
    }
};

inline int compressX(int x){
    return lower_bound(all_x.begin(), all_x.end(), x) - all_x.begin() + 1;
}

inline int compressY(int y){
    return lower_bound(all_y.begin(), all_y.end(), y) - all_y.begin() + 1;
}

node* root = new node();
```

```
int main(){

    scanf("%d %d %d\n", &r, &c, &t);
    for(int i = 1; i <= t; i++){
        scanf("%d ", &type[i]);
        if(type[i] == 1){
            scanf("%d %d %lld\n", &a1[i], &b1[i], &val[i]);
            a1[i]++, b1[i]++;
            all_x.push_back(a1[i]);
            all_y.push_back(b1[i]);
        }
        else{
            scanf("%d %d %d %d\n", &a1[i], &b1[i], &a2[i], &b2[i]);
            a1[i]++, b1[i]++, a2[i]++, b2[i]++;
            all_x.push_back(a1[i]), all_x.push_back(a2[i]);
            all_y.push_back(b1[i]), all_y.push_back(b2[i]);
        }
    }

    sort(all_x.begin(), all_x.end());
    sort(all_y.begin(), all_y.end());
    all_x.resize(unique(all_x.begin(), all_x.end()) - all_x.begin());
    all_y.resize(unique(all_y.begin(), all_y.end()) - all_y.begin());
    r = all_x.size(), c = all_y.size();

    for(int i = 1; i <= t; i++){
        if(type[i] == 1){
            a1[i] = compressX(a1[i]), b1[i] = compressY(b1[i]);
            root  = root -> update_x(1, r, a1[i], b1[i], val[i]);
        }
        else{
            a1[i] = compressX(a1[i]), a2[i] = compressX(a2[i]);
            b1[i] = compressY(b1[i]), b2[i] = compressY(b2[i]);
            printf("%lld\n", root -> query_x(1, r, a1[i], b1[i], a2[i], b2[i]));
        }
    }
}
```

1.2 Fenwick Tree

1.2.1 2D BIT

```
/*
   Copied from Alex Li's book code.
   Maintain a 2D array of numerical type, allowing for rectangular sub-matrices to
   be simultaneously incremented by arbitrary values (range update) and queries for
   the sum of rectangular sub-matrices (range query). This implementation uses
   std::map for coordinate compression, allowing for large indices to be accessed
   with efficient space complexity. That is, rows have valid indices from 0 to
   MAXR, inclusive, and columns have valid indices from 0 to MAXC, inclusive.
   - add(r, c, x) adds x to the value at index (r, c).
   - add(r1, c1, r2, c2, x) adds x to all indices in the rectangle with upper-left
   corner (r1, c1) and lower-right corner (r2, c2).
   - set(r, c, x) assigns x to the value at index (r, c).
   - sum(r, c) returns the sum of the rectangle with upper-left corner (0, 0) and
   lower-right corner (r, c).
   - sum(r1, c1, r2, c2) returns the sum of the rectangle with upper-left corner
   (r1, c1) and lower-right corner (r2, c2).
   - at(r, c) returns the value at index (r, c).
   Time Complexity:
```

```
   - O(log^2(MAXR)*log^2(MAXC)) per call to all member functions. If std::map is
   replaced with std::unordered_map, then the amortized running time will become
   O(log(MAXR)*log(MAXC)).
   Space Complexity:
   - O(n*log(MAXR)*log(MAXC)) for storage of the array elements, where n is the
   number of distinct indices that have been accessed across all of the
   operations so far.
   - O(1) auxiliary for all operations.
 */

#include <map>
#include <utility>

template<class T>
class fenwick_tree_2d {
    static const int MAXR = 1000000001;
    static const int MAXC = 1000000001;
    std::map<std::pair<int, int>, T> t1, t2, t3, t4;

    template<class Map>
        void add(Map &tree, int r, int c, const T &x) {
            for (int i = r + 1; i <= MAXR; i += i & -i) {
                for (int j = c + 1; j <= MAXC; j += j & -j) {
                    tree[std::make_pair(i, j)] += x;
                }
            }
        }

    void add_helper(int r, int c, const T &x) {
        add(t1, 0, 0, x);
        add(t1, 0, c, -x);
        add(t2, 0, c, x*c);
        add(t1, r, 0, -x);
        add(t3, r, 0, x*r);
        add(t1, r, c, x);
        add(t2, r, c, -x*c);
        add(t3, r, c, -x*r);
        add(t4, r, c, x*r*c);
    }

    public:
    void add(int r1, int c1, int r2, int c2, const T &x) {
        add_helper(r2 + 1, c2 + 1, x);
        add_helper(r1, c2 + 1, -x);
        add_helper(r2 + 1, c1, -x);
        add_helper(r1, c1, x);
    }

    void add(int r, int c, const T &x) {
        add(r, c, r, c, x);
    }

    void set(int r, int c, const T &x) {
        add(r, c, x - at(r, c));
    }

    T sum(int r, int c) {
        r++;
        c++;
        T s1 = 0, s2 = 0, s3 = 0, s4 = 0;
        for (int i = r; i > 0; i -= i & -i) {
            for (int j = c; j > 0; j -= j & -j) {
                const std::pair<int, int> ij(i, j);
                s1 += t1[ij];
                s2 += t2[ij];
                s3 += t3[ij];
                s4 += t4[ij];
            }
        }
        return s1*r*c + s2*r + s3*c + s4;
    }

    T sum(int r1, int c1, int r2, int c2) {
        return sum(r2, c2) + sum(r1 - 1, c1 - 1) -
            sum(r1 - 1, c2) - sum(r2, c1 - 1);
    }

    T at(int r, int c) {
        return sum(r, c, r, c);
    }
};

/*** Example Usage and Output:
Values:
5 6 0
3 5 5
0 5 14
 ***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
    fenwick_tree_2d<int> t;
    t.set(0, 0, 5);
    t.set(0, 1, 6);
    t.set(1, 0, 7);
    t.add(2, 2, 9);
    t.add(1, 0, -4);
    t.add(1, 1, 2, 2, 5);
    cout << "Values:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << t.at(i, j) << " ";
        }
        cout << endl;
    }
    assert(t.sum(0, 0, 0, 1) == 11);
    assert(t.sum(0, 0, 1, 0) == 8);
    assert(t.sum(1, 1, 2, 2) == 29);
    t.set(500000000, 500000000, 100);
    assert(t.sum(0, 0, 1000000000, 1000000000) == 143);
    return 0;
}
```

1.3 HLD Trick

1.3.1 HLD Trick

```
// Find number of pairs (u, v) such that A[u] * A[v] = A[lca(u, v)]
```

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

const int N = 1e5 + 50;

int n, arr[N], par[N];
vector < int > adj[N];
map < int, int > val[N];
long long ans = 0;

inline int root(int x){
    if(par[x] == x) return x;
    return par[x] = root(par[x]);
}

inline void unite(int u, int v, int target){
    u = root(u), v = root(v);
    if((int) val[u].size() < (int) val[v].size()){
        for(map < int, int > :: iterator it = val[u].begin(); it != val[u].end(); it++){
            int cur = (*it).first;
            if(target % cur == 0) ans += ((*it).second * 1LL * val[v][target / cur]);
        }
        for(map < int, int > :: iterator it = val[u].begin(); it != val[u].end(); it++){
            int cur = (*it).first;
            val[v][cur] += (*it).second;
        }
        val[u].clear();
        par[u] = v;
    }
    else{
        for(map < int, int > :: iterator it = val[v].begin(); it != val[v].end(); it++){
            int cur = (*it).first;
            if(target % cur == 0) ans += ((*it).second * 1LL * val[u][target / cur]);
        }
        for(map < int, int > :: iterator it = val[v].begin(); it != val[v].end(); it++){
            int cur = (*it).first;
            val[u][cur] += (*it).second;
        }
        val[v].clear();
        par[v] = u;
    }
}

inline void dfs(int u, int p){
    val[u][arr[u]]++;
    for(int i = 0; i < (int) adj[u].size(); i++){
        int v = adj[u][i];
        if(v == p) continue;
        dfs(v, u);
        unite(u, v, arr[u]);
    }
}

int main(){
    freopen("inp.in", "r", stdin);
    scanf("%d", &n);
    for(int i = 1; i < n; i++){
        int u, v;
        scanf("%d %d", &u, &v);
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    for(int i = 1; i <= n; i++){
        scanf("%d", arr + i);
        par[i] = i;
    }
    dfs(1, -1);
    printf("%lld\n", ans);
}
```

1.4 LIS Tricks

1.4.1 ZIPLine LIS

```
// Zipline Problem on Codeforces
// LIS after modifying one element of the array (in the queries)
// Gives us information on which elements exist in every LIS, and which exist in some,
// and which exist in none.

#include <bits/stdc++.h>
#define rf freopen("inp.in", "r", stdin)
using namespace std;

const int mx = 400005;

int n, q;
int arr[mx], f[mx], g[mx], pref[mx], suf[mx];
int bit1[mx], bit2[mx], cnt[mx], lis = 1;
vector < pair < int , pair < int, int > > > event;
pair < int, int > query[mx];
vector < int > elements;
int comp[mx];

void update1(int x, int v){
    for(int i = x; i < mx; i += i & (-i))
        bit1[i] = max(bit1[i], v);
}

int query1(int x){
    int res = 0;
    for(int i = x; i; i -= i & (-i))
        res = max(res, bit1[i]);
    return res;
}

void update2(int x, int v){
    for(int i = x; i; i -= i & (-i))
        bit2[i] = max(bit2[i], v);
}

int query2(int x){
    int res = 0;
    for(int i = x; i < mx; i += i & (-i))
        res = max(res, bit2[i]);
    return res;
}

int main(){

    scanf("%d %d\n", &n, &q);
    for(int i = 1; i <= n; i++){
```

```
    scanf("%d ", &arr[i]);
    elements.push_back(arr[i]);
    event.push_back(make_pair(arr[i], make_pair(0, i)));
}

sort(elements.begin(), elements.end());
elements.resize(unique(elements.begin(), elements.end()) - elements.begin());

for(int i = 1; i <= n; i++){
    comp[i] = lower_bound(elements.begin(), elements.end(), arr[i]) - elements.begin() + 1;
}

for(int i = 1; i <= q; i++){
    int pos, val;
    scanf("%d %d\n", &pos, &val);
    event.push_back(make_pair(val, make_pair(-i, pos)));
    query[i] = (make_pair(pos, val));
}

sort(event.begin(), event.end());

for(int i = 1; i <= n; i++){
    f[i] = query1(comp[i] - 1) + 1;
    update1(comp[i], f[i]);
    lis = max(lis, f[i]);
}

for(int i = n; i >= 1; i--){
    g[i] = query2(comp[i] + 1) + 1;
    update2(comp[i], g[i]);
}

memset(bit1, 0, sizeof bit1);
memset(bit2, 0, sizeof bit2);

for(int i = 0 ; i < event.size(); i++){
    int type = event[i].second.first;
    int pos  = event[i].second.second;
    if(type == 0){ // Array element
        update1(pos, f[pos]);
    }
    else{ // Query element
        pref[-type] = query1(pos - 1) + 1;
    }
    event[i].second.first = (-type);
}

sort(event.begin(), event.end());
reverse(event.begin(), event.end());

for(int i = 0 ; i < event.size(); i++){
    int type = event[i].second.first;
    int pos  = event[i].second.second;
    if(type == 0){ // Array element
        update2(pos, g[pos]);
    }
    else{ // Query element
        suf[type] = query2(pos + 1) + 1;
    }
}

for(int i = 1; i <= n; i++){
        if(f[i] + g[i] - 1 == lis){
            cnt[f[i]]++;
        }
    }

    for(int i = 1; i <= q; i++){
        int pos = query[i].first, res = 1;
        res = max(res, pref[i] + suf[i] - 1); // lis including (i)
        if(f[pos] + g[pos] - 1 == lis && cnt[f[pos]] == 1)
            res = max(res, lis - 1); // lis decreases
        else
            res = max(res, lis);  // lis constant
        printf("%d\n", res);
    }

}
```

1.5 Link Cut Tree

1.5.1 Link Cut Tree

```
/*
   All credit to Alex Li.

   Maintain a forest of trees with values associated with its nodes, while
   supporting both dynamic queries and dynamic updates of all values on any path
   between two nodes in a given tree. In addition, support testing of whether two
   nodes are connected in the forest, as well as the merging and spliting of trees
   by adding or removing specific edges. Link/cut forests divide each of its trees
   into vertex-disjoint paths, each represented by a splay tree.
   The query operation is defined by an associative join_values() function which
   satisfies join_values(x, join_values(y, z)) = join_values(join_values(x, y), z)
   for all values x, y, and z in the forest. The default code below assumes a
   numerical forest type, defining queries for the "min" of the target range.
   Another possible query operation is "sum", in which case the join_values()
   function should be defined to return "a + b".
   The update operation is defined by the join_value_with_delta() and join_deltas()
   functions, which determines the change made to values. These must satisfy:
   - join_deltas(d1, join_deltas(d2, d3)) = join_deltas(join_deltas(d1, d2), d3).
   - join_value_with_delta(join_values(v, ...(m times)..., v), d, m)) should be
   equal to join_values(join_value_with_delta(v, d, 1), ...(m times)).
   - if a sequence d_1, ..., d_m of deltas is used to update a value v, then
   join_value_with_delta(v, join_deltas(d_1, ..., d_m), 1) should be equivalent
   to m sequential calls to join_value_with_delta(v, d_i, 1) for i = 1..m.
   The default code below defines updates that "set" a path's nodes to a new value.
   Another possible update operation is "increment", in which case
   join_value_with_delta(v, d, len) should be defined to return "v + d*len" and
   join_deltas(d1, d2) should be defined to return "d1 + d2".
   - link_cut_forest() constructs an empty forest with no trees.
   - size() returns the number of nodes in the forest.
   - trees() returns the number of trees in the forest.
   - make_root(i, v) creates a new tree in the forest consisting of a single node
   labeled with the integer i and value initialized to v.
   - is_connected(a, b) returns whether nodes a and b are connected.
   - link(a, b) adds an edge between the nodes a and b, both of which must exist
   and not be connected.
   - cut(a, b) removes the edge between the nodes a and b, both of which must
   exist and be connected.
   - query(a, b) returns the result of join_values() applied to all values on the
   path from the node a to node b.
```

```
  - update(a, b, d) modifies all the values on the path from node a to node b by
  respectively joining them with d using join_value_with_delta().
  Time Complexity:
  - O(1) per call to the constructor, size(), and trees().
  - O(log n) amortized per call to all other operations, where n is the number of
  nodes.
  Space Complexity:
  - O(n) for storage of the forest, where n is the number of nodes.
  - O(1) auxiliary for all operations.
 */

#include <algorithm>
#include <cstdlib>
#include <map>
#include <stdexcept>

template<class T>
class link_cut_forest {
    static T join_values(const T &a, const T &b) {
        return std::min(a, b);
    }

    static T join_value_with_delta(const T &v, const T &d, int len) {
        return d;
    }

    static T join_deltas(const T &d1, const T &d2) {
        return d2;  // For "set" updates, the more recent delta prevails.
    }

    struct node_t {
        T value, subtree_value, delta;
        int size;
        bool rev, pending;
        node_t *left, *right, *parent;

        node_t(const T &v)
            : value(v), subtree_value(v), size(1), rev(false), pending(false),
            left(NULL), right(NULL), parent(NULL) {}

        inline bool is_root() const {
            return parent == NULL || (parent->left != this && parent->right != this);
        }

        inline T get_subtree_value() const {
            return pending ? join_value_with_delta(subtree_value, delta, size)
                : subtree_value;
        }

        void push() {
            if (rev) {
                rev = false;
                std::swap(left, right);
                if (left != NULL) {
                    left->rev = !left->rev;
                }
                if (right != NULL) {
                    right->rev = !right->rev;
                }
            }
            if (pending) {
                value = join_value_with_delta(value, delta, 1);
                subtree_value = join_value_with_delta(subtree_value, delta, size);
                if (left != NULL) {
                    left->delta = left->pending ? join_deltas(left->delta, delta) : delta;
                    left->pending = true;
                }
                if (right != NULL) {
                    right->delta = right->pending ? join_deltas(right->delta, delta)
                        : delta;
                    right->pending = true;
                }
                pending = false;
            }
        }

        void update() {
            size = 1;
            subtree_value = value;
            if (left != NULL) {
                subtree_value = join_values(subtree_value, left->get_subtree_value());
                size += left->size;
            }
            if (right != NULL) {
                subtree_value = join_values(subtree_value, right->get_subtree_value());
                size += right->size;
            }
        }
    };

    int num_trees;
    std::map<int, node_t*> nodes;

    static void connect(node_t *child, node_t *parent, bool is_left) {
        if (child != NULL) {
            child->parent = parent;
        }
        if (is_left) {
            parent->left = child;
        } else {
            parent->right = child;
        }
    }

    static void rotate(node_t *n) {
        node_t *parent = n->parent, *grandparent = parent->parent;
        bool parent_is_root = parent->is_root(), is_left = (n == parent->left);
        connect(is_left ? n->right : n->left, parent, is_left);
        connect(parent, n, !is_left);
        if (parent_is_root) {
            if (n != NULL) {
                n->parent = grandparent;
            }
        } else {
            connect(n, grandparent, parent == grandparent->left);
        }
        parent->update();
    }

    static void splay(node_t *n) {
        while (!n->is_root()) {
            node_t *parent = n->parent, *grandparent = parent->parent;
            if (!parent->is_root()) {
                grandparent->push();
            }
            parent->push();
            n->push();
            if (!parent->is_root()) {
                if ((n == parent->left) == (parent == grandparent->left)) {
                    rotate(parent);
                } else {
                    rotate(n);
                }
            }
            rotate(n);
        }
        n->push();
        n->update();
    }

    static node_t* expose(node_t *n) {
        node_t *prev = NULL;
        for (node_t *curr = n; curr != NULL; curr = curr->parent) {
            splay(curr);
            curr->left = prev;
            prev = curr;
        }
        splay(n);
        return prev;
    }

    // Helper variables.
    node_t *u, *v;

    void get_uv(int a, int b) {
        typename std::map<int, node_t*>::iterator it1, it2;
        it1 = nodes.find(a);
        it2 = nodes.find(b);
        if (it1 == nodes.end() || it2 == nodes.end()) {
            throw std::runtime_error("Queried node ID does not exist in forest.");
        }
        u = it1->second;
        v = it2->second;
    }

    public:
    link_cut_forest() : num_trees(0) {}

    ~link_cut_forest() {
        typename std::map<int, node_t*>::iterator it;
        for (it = nodes.begin(); it != nodes.end(); ++it) {
            delete it->second;
        }
    }

    int size() const {
        return nodes.size();
    }

    int trees() const {
        return num_trees;
    }

    void make_root(int i, const T &v = T()) {
        if (nodes.find(i) != nodes.end()) {
            throw std::runtime_error("Cannot make a root with an existing ID.");
        }
        node_t *n = new node_t(v);
        expose(n);
        n->rev = !n->rev;
        nodes[i] = n;
        num_trees++;
    }

    bool is_connected(int a, int b) {
        get_uv(a, b);
        if (a == b) {
            return true;
        }
        expose(u);
        expose(v);
        return u->parent != NULL;
    }

    void link(int a, int b) {
        if (is_connected(a, b)) {
            throw std::runtime_error("Cannot link nodes that are already connected.");
        }
        get_uv(a, b);
        expose(u);
        u->rev = !u->rev;
        u->parent = v;
        num_trees--;
    }

    void cut(int a, int b) {
        get_uv(a, b);
        expose(u);
        u->rev = !u->rev;
        expose(v);
        if (v->right != u || u->left != NULL) {
            throw std::runtime_error("Cannot cut edge that does not exist.");
        }
        v->right->parent = NULL;
        v->right = NULL;
        num_trees++;
    }

    T query(int a, int b) {
        if (!is_connected(a, b)) {
            throw std::runtime_error("Cannot query nodes that are not connected.");
        }
        get_uv(a, b);
        expose(u);
        u->rev = !u->rev;
        expose(v);
        return v->get_subtree_value();
    }

    void update(int a, int b, const T &d) {
        if (!is_connected(a, b)) {
            throw std::runtime_error("Cannot update nodes that are not connected.");
        }
        get_uv(a, b);
        expose(u);
        u->rev = !u->rev;
        expose(v);
        v->delta = v->pending ? join_deltas(v->delta, d) : d;
```

```
        v->pending = true;
    }
};

/*** Example Usage ***/

#include <cassert>
using namespace std;

int main() {
    link_cut_forest<int> lcf;
    lcf.make_root(0, 10);
    lcf.make_root(1, 40);
    lcf.make_root(2, 20);
    lcf.make_root(3, 10);
    lcf.make_root(4, 30);
    assert(lcf.size() == 5);
    assert(lcf.trees() == 5);
    lcf.link(0, 1);
    lcf.link(1, 2);
    lcf.link(2, 3);
    lcf.link(2, 4);
    assert(lcf.trees() == 1);

    // v=10      v=40      v=20      v=10
    //  0---------1---------2---------3
    //                      |
    //                      ----------4
    //                               v=30
    assert(lcf.query(1, 4) == 20);
    lcf.update(1, 1, 100);
    lcf.update(2, 4, 100);

    // v=10     v=100     v=100      v=10
    //  0---------1---------2---------3
    //                      |
    //                      ----------4
    //                              v=100
    assert(lcf.query(4, 4) == 100);
    assert(lcf.query(0, 4) == 10);
    assert(lcf.query(3, 4) == 10);
    lcf.cut(1, 2);

    // v=10     v=100     v=100      v=0
    //  0---------1         2---------3
    //                      |
    //                      ----------4
    //                              v=100
    assert(lcf.trees() == 2);
    assert(!lcf.is_connected(1, 2));
    assert(!lcf.is_connected(0, 4));
    assert(lcf.is_connected(2, 3));
    return 0;
}
```

1.6 Merge Sort Tree

1.6.1 Merge Sort Tree

```
// SEGSUMQ: Codechef Snackdown Elimination 2016
```

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

const int N = 1e5 + 5;
const double INF = 1e18;

int n, q;
long long a[N], b[N];
vector < pair < long long, long long > > tree[2][N * 4];

// Sort by a[i] / b[i]
inline bool compare(pair < long long, long long > x, pair < long long, long long > y){
    double valx, valy;
    if(x.second != 0) valx  = x.first * 1.0 / x.second;
    else valx = INF;
    if(y.second != 0) valy  = y.first * 1.0 / y.second;
    else valy = INF;
    return (valx >= valy);
}

inline pair < long long, long long > query(int node, int l, int r, int qs, int qe, long long c, long long d){
    if(l > qe || r < qs) return {0, 0};
    if(l >= qs && r <= qe){
        int l = 0, r = tree[0][node].size() - 1;
        int idx = 0;
        while(l <= r){
            int mid = (l + r) / 2;
            if(compare(tree[0][node][mid], {c, d})){
                idx = mid;
                l = mid + 1;
            }
            else r = mid - 1;
        }
        if(compare(tree[0][node][idx], {c, d})) return tree[1][node][idx];
        return {0, 0};
    }
    int mid = (l + r) >> 1;
    pair < long long, long long > x = query(node * 2, l, mid, qs, qe, c, d);
    pair < long long, long long > y = query(node * 2 + 1, mid + 1, r, qs, qe, c, d);
    return {x.first + y.first, x.second + y.second};
}


inline void build(int node, int l, int r){
    if(l == r){
        tree[0][node].push_back({a[l], b[l]});
        tree[1][node].push_back({a[l], b[l]});
        return;
    }
    int mid = (l + r) >> 1;
    build(node * 2, l, mid);
    build(node * 2 + 1, mid + 1, r);
    tree[0][node].resize(tree[0][node * 2].size() + tree[0][node * 2 + 1].size());
    merge(tree[0][node * 2].begin(), tree[0][node * 2].end(), tree[0][node * 2 + 1].begin(),
            tree[0][node * 2 + 1].end(), tree[0][node].begin(), compare);
    pair < long long, long long > prefix_sum = {0, 0};
    for(int i = 0; i < (int) tree[0][node].size(); i++){
        prefix_sum.first  += tree[0][node][i].first;
        prefix_sum.second += tree[0][node][i].second;
        tree[1][node].push_back(prefix_sum);
    }
}
```

```
int main(){
    scanf("%d", &n);
    for(int i = 1; i <= n; i++) scanf("%lld", a + i);
    for(int i = 1; i <= n; i++) scanf("%lld", b + i);
    build(1, 1, n);
    scanf("%d", &q);
    while(q--){
        long long l, r, c, d;
        scanf("%lld %lld %lld %lld", &l, &r, &c, &d);
        pair < long long, long long > res = query(1, 1, n, l, r, d, c);
        fflush(stdout);
        printf("%lld\n", res.first * 1LL * c - res.second * 1LL * d);
        fflush(stdout);
    }
}
```

1.7 Persistent Segment Trees

1.7.1 2D PST

```
/*
   l r k : Let S denote the sorted (in increasing order) set of elements of array A with
   its indices between l and r. Note that set S contains distinct elements.
   You need to find kth number in it.
   If such a number does not exist, i.e. the S has less than k elements, output -1.
 */

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

const int N = 100005;
const int L = 1;
const int R = 100000;

int n, q, arr[N], orig[N], latest[N];
vector < int > values;

struct node_y{ // Contains latest occurence positions of values.
    node_y *lc, *rc;
    int val;
    node_y(node_y *l = NULL, node_y *r = NULL, int v = 0){
        lc = l, rc = r, val = v;
    }
    inline void create(node_y* &x){
        if(!x) x = new node_y();
    }
    inline int sum(node_y *x){
        return (x) ? (x -> val) : (0);
    }
    inline node_y *update_y(int l, int r, int pos, int add){
        node_y *nw = new node_y();
        if(l == r){
            nw -> val = (val + add);
            return nw;
        }
        int mid = (l + r) >> 1;
        if(mid >= pos){
            nw -> rc = rc;
            create(lc);
            nw -> lc = lc -> update_y(l, mid, pos, add);
        }
        else{
            nw -> lc = lc;
            create(rc);
            nw -> rc = rc -> update_y(mid + 1, r, pos, add);
        }
        nw -> val = sum(nw -> lc) + sum(nw -> rc);
        return nw;
    }
    inline int query_y(int l, int r, int qs, int qe){
        if(l > qe || r < qs) return 0;
        if(l >= qs && r <= qe) return val;
        int mid = (l + r) >> 1, res = 0;
        if(lc) res += lc -> query_y(l, mid, qs, qe);
        if(rc) res += rc -> query_y(mid + 1, r, qs, qe);
        return res;
    }
};

/*
   It's a persistent segment tree in which every node is also another
   persistent segment tree. The base tree is indexed by values of nodes.
   Thus, in a node corresponding to range [min, max], we can query
   "how many values in [min, max] have their latest occurence >= x" by
   querying the 2nd-dimenion-persistent-segment-tree. This allows us
   to binary search the answer.
 */

struct node_x{ // Contains the numbers (We will binary search to find kth element)
    node_x *lc, *rc;
    node_y *outer;
    int val;
    node_x(node_x *l = NULL, node_x *r = NULL, int v = 0){
        lc = l, rc = r, val = v;
        outer = new node_y();
    }
    inline void create(node_x* &x){
        if(!x) x = new node_x();
    }
    inline node_x *update_x(int l, int r, int value, int prev_occ, int curr_occ){
        node_x *nw = new node_x();
        if(!outer) outer = new node_y();
        if(prev_occ){
            nw -> outer = outer -> update_y(1, n, prev_occ, -1);
            nw -> outer = nw -> outer -> update_y(1, n, curr_occ, 1);
        }
        else{
            nw -> outer = outer -> update_y(1, n, curr_occ, 1);
        }
        if(l == r) return nw;
        int mid = (l + r) >> 1;
        if(mid >= value){
            nw -> rc = rc;
            create(lc);
            nw -> lc = lc -> update_x(l, mid, value, prev_occ, curr_occ);
        }
        else{
            nw -> lc = lc;
            create(rc);
            nw -> rc = rc -> update_x(mid + 1, r, value, prev_occ, curr_occ);
        }
        return nw;
    }
    inline int query_x(int l, int r, int qs, int qe, int k){
        if(l == r) return l;
        if(l == L && r == R){
            if(!outer) return -1;
            if(outer -> query_y(1, n, qs, qe) < k) return -1;
        }
        int mid = (l + r) >> 1, goLeft = 0;
        if(lc && lc -> outer) goLeft = lc -> outer -> query_y(1, n, qs, qe);
        if(goLeft >= k) return lc -> query_x(l, mid, qs, qe, k);
        else return rc -> query_x(mid + 1, r, qs, qe, k - goLeft);
    }
};

node_x *root[N];

int main(){
    scanf("%d %d", &n, &q);
    for(int i = 1; i <= n; i++){
        scanf("%d", arr + i);
        values.push_back(arr[i]);
    }

    sort(values.begin(), values.end());
    values.resize(unique(values.begin(), values.end()) - values.begin());

    for(int i = 1; i <= n; i++){
        int copy = arr[i];
        arr[i] = lower_bound(values.begin(), values.end(), arr[i]) - values.begin() + 1;
        orig[arr[i]] = copy;
    }

    root[0] = new node_x();

    for(int i = 1; i <= n; i++){
        int prev_occ = latest[arr[i]];
        latest[arr[i]] = i;
        root[i] = root[i - 1] -> update_x(L, R, arr[i], prev_occ, i);
    }

    int ans = 0;
    while(q--){
        int a, b, c, d, k;
        scanf("%d %d %d %d %d", &a, &b, &c, &d, &k);

        int l = ((a * 1LL * max(0, ans) + b) % n) + 1;
        int r = ((c * 1LL * max(0, ans) + d) % n) + 1;
        if(l > r) swap(l, r);

        ans = root[r] -> query_x(L, R, l, r, k);
        if(ans != -1) ans = orig[ans];
        printf("%d\n", ans);
    }
}
```

1.7.2 Non Pointer PST

```
// Print minimal x such that x occurs in the interval [l, r] strictly more than (r - l + 1) / k times.
```

```cpp
// Complexity: O(Q * max_k * log(n))
#include "bits/stdc++.h"
using namespace std;

// Modify to handle values
const int N = 12345678;

// 0 = null

int n, q, ticks;
int root[N], lc[N], rc[N], sum[N];

inline int insert(int old_root, int l, int r, int pos, int val) {
    int cur_root = ++ticks;
    lc[cur_root] = lc[old_root];
    rc[cur_root] = rc[old_root];
    sum[cur_root] = sum[old_root] + val;
    if (l == r) {
        return cur_root;
    }
    int mid = (l + r) >> 1;
    if (mid >= pos) {
        lc[cur_root] = insert(lc[old_root], l, mid, pos, val);
    } else {
        rc[cur_root] = insert(rc[old_root], mid + 1, r, pos, val);
    }
    return cur_root;
}

inline int query(int lid, int rid, int l, int r, int min_freq) {
    int num_values_in_this_range = sum[rid] - sum[lid];
    if (num_values_in_this_range < min_freq) {
        return -1;
    }
    if (l == r) {
        return l;
    }
    int mid = (l + r) >> 1;
    int lol = query(lc[lid], lc[rid], l, mid, min_freq);
    if (lol != -1) {
        return lol;
    }
    return query(rc[lid], rc[rid], mid + 1, r, min_freq);
}

int main() {
    freopen ("inp.in", "r", stdin);
    ios :: sync_with_stdio(false);
    cin >> n >> q;
    for (int i = 1; i <= n; i++) {
        int x; cin >> x;
        root[i] = insert(root[i - 1], 1, n, x, 1);
    }
    while (q--) {
        int l, r, k; cin >> l >> r >> k;
        int freq = (r - l + 1) / k + 1;
        cout << query(root[l - 1], root[r], 1, n, freq) << endl;
    }
}
```

1.7.3 Pointer PST

```
/*
   Prints sum of K maximum sum subarrays, each of L <= length <= R
   Array has negative elements as well.

   Add f[i]th best subarray starting at index (i) of valid length for each (i)
   into a priority queue. Initially, let f[i] = 1 for all (i).
   Pop the best value from the priority queue k times, increment f[i] each time
   and add a new value to it after each pop.
 */

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

const int MAX = 500005;
const int INF = 1000000000;

struct node{
    node *lc, *rc;
    int val;
    node(node *x = NULL, node *y = NULL, int v = 0){
        lc = x, rc = y, val = v;
    }
    inline void create(node *&x){
        if(!x) x = new node();
    }
    inline int sum(node *x){
        return (x) ? (x -> val) : (0);
    }
    inline node *insert(int l, int r, int value){
        node *nw = new node();
        if(l == r){
            nw -> val = val + 1;
            return nw;
        }
        int mid = (l + r) >> 1;
        if(mid >= value){
            nw -> rc = rc;
            create(lc);
            nw -> lc = lc -> insert(l, mid, value);
        }
        else{
            nw -> lc = lc;
            create(rc);
            nw -> rc = rc -> insert(mid + 1, r, value);
        }
        nw -> val = sum(nw -> lc) + sum(nw -> rc);
        return nw;
    }
    inline int query(node *r1, node *r2, int l, int r, int k){
        if(l == r) return r;
        int goRight = sum(r1 -> rc) - sum(r2 -> rc);
        int mid = (l + r) >> 1;
        if(goRight >= k){
            create(r1 -> rc), create(r2 -> rc);
            return query(r1 -> rc, r2 -> rc, mid + 1, r, k);
        }
        else{
            create(r1 -> lc), create(r2 -> lc);
            return query(r1 -> lc, r2 -> lc, l, mid, k - goRight);
        }
    }
};

node *root[MAX], *dummy;
map < int, int > compress;
int n, k, l, r, lim;
int arr[MAX], f[MAX], original[MAX];

inline int get(int i, int j){
    if(i + l - 1 > n || j > r - l + 1) return -INF;
    return original[dummy -> query(root[min(i + r - 1, n)], root[i + l - 2], 1, lim, j)] - arr[i - 1];
}

int main(){
    scanf("%d %d %d %d", &n, &k, &l, &r);
    compress[-INF];
    for(int i = 1; i <= n; i++){
        scanf("%d", arr + i);
        arr[i] += arr[i - 1];
        compress[arr[i]];
    }
    for(auto &it : compress) it.second = ++lim;
    for(auto  it : compress) original[it.second] = it.first;
    root[0] = dummy = new node();
    for(int i = 1; i <= n; i++){
        root[i] = root[i - 1] -> insert(1, lim, compress[arr[i]]);
    }
    priority_queue < pair < int, int > > sums;
    for(int i = 1; i <= n; i++){
        sums.push({get(i, ++f[i]), i});
    }
    long long res = 0;
    while(k--){
        res += (sums.top().first);
        sums.push({get(sums.top().second, ++f[sums.top().second]), sums.top().second});
        sums.pop();
    }
    printf("%lld\n", res);
}
```

1.8 SQRT Decomposition

1.8.1 Mos on Trees

```
// You are given a tree with N nodes. The tree nodes are numbered from 1 to N. Each node has an integer weight.
// We will ask you to perform the following operation:
// u v : Ask for how many different integers that represent the weight of nodes there are on the path from u to v.

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

const int MAXN = 40005;
const int MAXM = 100005;
const int LN = 19;

int N, M, K, cur, A[MAXN], LVL[MAXN], DP[LN][MAXN];
int BL[MAXN << 1], ID[MAXN << 1], VAL[MAXN], ANS[MAXM];
int d[MAXN], l[MAXN], r[MAXN];
bool VIS[MAXN];
```

```
vector < int > adjList[MAXN];

struct query{
    int id, l, r, lc;
    bool operator < (const query& rhs){
        return (BL[l] == BL[rhs.l]) ? (r < rhs.r) : (BL[l] < BL[rhs.l]);
    }
}Q[MAXM];

// Set up Stuff
void dfs(int u, int par){
    l[u] = ++cur;
    ID[cur] = u;
    for (int i = 1; i < LN; i++) DP[i][u] = DP[i - 1][DP[i - 1][u]];
    for (int i = 0; i < adjList[u].size(); i++){
        int v = adjList[u][i];
        if (v == par) continue;
        LVL[v] = LVL[u] + 1;
        DP[0][v] = u;
        dfs(v, u);
    }
    r[u] = ++cur; ID[cur] = u;
}

// Function returns lca of (u) and (v)
inline int lca(int u, int v){
    if (LVL[u] > LVL[v]) swap(u, v);
    for (int i = LN - 1; i >= 0; i--)
        if (LVL[v] - (1 << i) >= LVL[u]) v = DP[i][v];
    if (u == v) return u;
    for (int i = LN - 1; i >= 0; i--){
        if (DP[i][u] != DP[i][v]){
            u = DP[i][u];
            v = DP[i][v];
        }
    }
    return DP[0][u];
}

inline void check(int x, int& res){
    // If (x) occurs twice, then don't consider it's value
    if ( (VIS[x]) and (--VAL[A[x]] == 0) ) res--;
    else if ( (!VIS[x]) and (VAL[A[x]]++ == 0) ) res++;
    VIS[x] ^= 1;
}

void compute(){

    // Perform standard Mo's Algorithm
    int curL = Q[0].l, curR = Q[0].l - 1, res = 0;

    for (int i = 0; i < M; i++){

        while (curL < Q[i].l) check(ID[curL++], res);
        while (curL > Q[i].l) check(ID[--curL], res);
        while (curR < Q[i].r) check(ID[++curR], res);
        while (curR > Q[i].r) check(ID[curR--], res);

        int u = ID[curL], v = ID[curR];

        // Case 2
        if (Q[i].lc != u and Q[i].lc != v) check(Q[i].lc, res);
        ANS[Q[i].id] = res;

        // Case 2
        if (Q[i].lc != u and Q[i].lc != v) check(Q[i].lc, res);
    }

    for (int i = 0; i < M; i++) printf("%d\n", ANS[i]);
}

int main(){

    int u, v, x;

    while (scanf("%d %d", &N, &M) != EOF){

        // Cleanup
        cur = 0;
        memset(VIS, 0, sizeof(VIS));
        memset(VAL, 0, sizeof(VAL));
        for (int i = 1; i <= N; i++) adjList[i].clear();

        // Inputting Values
        for (int i = 1; i <= N; i++) scanf("%d", &A[i]);
        memcpy(d + 1, A + 1, sizeof(int) * N);

        // Compressing Coordinates
        sort(d + 1, d + N + 1);
        K = unique(d + 1, d + N + 1) - d - 1;
        for (int i = 1; i <= N; i++) A[i] = lower_bound(d + 1, d + K + 1, A[i]) - d;

        // Inputting Tree
        for (int i = 1; i < N; i++){
            scanf("%d %d", &u, &v);
            adjList[u].push_back(v);
            adjList[v].push_back(u);
        }

        // Preprocess
        DP[0][1] = 1;
        dfs(1, -1);
        int size = sqrt(cur);

        for (int i = 1; i <= cur; i++) BL[i] = (i - 1) / size + 1;

        for (int i = 0; i < M; i++){
            scanf("%d %d", &u, &v);
            Q[i].lc = lca(u, v);
            if (l[u] > l[v]) swap(u, v);
            if (Q[i].lc == u) Q[i].l = l[u], Q[i].r = l[v];
            else Q[i].l = r[u], Q[i].r = l[v];
            Q[i].id = i;
        }

        sort(Q, Q + M);
        compute();
    }
}
```

1.8.2 Mos with Updates

```
// Primitive Queries - DISTNUM3
#include "bits/stdc++.h"
using namespace std;

const int N = 100000 + 50;
const int LN = 17;

int n, q, blk, ticks, qid, uid, res;
int tin[N], tout[N], depth[N], block_id[N * 2];
int color[N], original_color[N], order[2 * N];
int ans[N], vis[N], freq[2 * N], last[N];
int anc[LN][N];
vector < int > all_colors;
vector < int > adj[N];

struct query {
    int l, r, lc, t, id;
    query(int _l = 0, int _r = 0, int _lc = 0, int _t = 0, int _id = 0) {
        l = _l;
        r = _r;
        lc = _lc;
        t = _t;
        id = _id;
    }
    inline friend bool operator < (query a, query b) {
        if (block_id[a.l] != block_id[b.l]) {
            return (block_id[a.l] < block_id[b.l]);
        } else if (block_id[a.r] != block_id[b.r]) {
            return (block_id[a.r] < block_id[b.r]);
        } else if (a.t != b.t) {
            return (a.t < b.t);
        } else {
            return (a.id < b.id);
        }
    }
}queries[N];

struct update {
    int node, old_color, new_color;
    update(int _node = 0, int _old_color = 0, int _new_color = 0) {
        node = _node;
        old_color = _old_color;
        new_color = _new_color;
    }
}updates[N];

inline int compress(int x) {
    return lower_bound(all_colors.begin(), all_colors.end(), x) - all_colors.begin();
}

inline void dfs(int u, int p) {
    tin[u] = ++ticks;
    order[ticks] = u;
    anc[0][u] = p;
    for (int i = 1; i < LN; i++) {
        anc[i][u] = anc[i - 1][anc[i - 1][u]];
    }
    for (int v : adj[u]) {
        if (v == p) {
            continue;
        }
        depth[v] = depth[u] + 1;
        dfs(v, u);
    }
    tout[u] = ++ticks;
    order[ticks] = u;
}

inline int lca(int x, int y) {
    if (depth[x] < depth[y]) {
        swap(x, y);
    }
    for (int i = LN - 1; i >= 0; i--) {
        if (depth[x] - (1 << i) >= depth[y]) {
            x = anc[i][x];
        }
    }
    if (x == y) {
        return x;
    }
    for (int i = LN - 1; i >= 0; i--) {
        if (anc[i][x] != anc[i][y]) {
            x = anc[i][x];
            y = anc[i][y];
        }
    }
    return anc[0][x];
}

inline void visit(int timer) {
    int node = order[timer];
    if (vis[node]) {
        if (freq[color[node]] == 1) {
            res--;
        }
        freq[color[node]]--;
    } else {
        if (freq[color[node]] == 0) {
            res++;
        }
        freq[color[node]]++;
    }
    vis[node] ^= 1;
}

inline void change_color(int u, int x) {
    if (vis[u]) {
        visit(tin[u]);
        color[u] = x;
        visit(tin[u]);
    } else {
        color[u] = x;
    }
}

int main() {
    scanf("%d %d", &n, &q);
    blk = (cbrt(2 * n + 10)) * (cbrt(2 * n + 10));
    for (int i = 1; i <= n; i++) {
        scanf("%d", &original_color[i]);
        all_colors.push_back(original_color[i]);
        last[i] = original_color[i];
    }
    for (int i = 1; i <= 2 * n; i++) {
        block_id[i] = i / blk + 1;
    }
    for (int i = 1; i < n; i++) {
        int foo, bar; scanf("%d %d", &foo, &bar);
        adj[foo].push_back(bar);
        adj[bar].push_back(foo);
    }
    dfs(1, 1);
    for (int i = 1; i <= q; i++) {
        int type; scanf("%d", &type);
        if (type == 1) {
            int u, v; scanf("%d %d", &u, &v);
            if (tin[u] > tin[v]) {
                swap(u, v);
            }
            int lc = lca(u, v);
            if (lc == u) {
                queries[1 + qid] = query(tin[u], tin[v], 0, uid, 1 + qid);
            } else {
                queries[1 + qid] = query(tout[u], tin[v], lc, uid, 1 + qid);
            }
            ++qid;
        } else {
            int v, y; scanf("%d %d", &v, &y);
            updates[++uid] = update(v, last[v], y);
            all_colors.push_back(y);
            last[v] = y;
        }
    }
    // Coordinate Compression
    sort(all_colors.begin(), all_colors.end());
    all_colors.resize(unique(all_colors.begin(), all_colors.end()) - all_colors.begin());
    for (int i = 1; i <= n; i++) {
        color[i] = compress(original_color[i]);
    }
    for (int i = 1; i <= uid; i++) {
        updates[i].old_color = compress(updates[i].old_color);
        updates[i].new_color = compress(updates[i].new_color);
    }
    // End of Coordinate Compression
    sort(queries + 1, queries + 1 + qid);
    int ql = 1, qr = 0, upd = 0;
    for (int i = 1; i <= qid; i++) {
        while (upd < queries[i].t) {
            ++upd;
            change_color(updates[upd].node, updates[upd].new_color);
        }
        while (upd > queries[i].t) {
            change_color(updates[upd].node, updates[upd].old_color);
            --upd;
        }
        while (ql < queries[i].l) {
            visit(ql++);
        }
        while (ql > queries[i].l) {
            visit(--ql);
        }
        while (qr < queries[i].r) {
            visit(++qr);
        }
        while (qr > queries[i].r) {
            visit(qr--);
        }
        if (queries[i].lc) {
            visit(tin[queries[i].lc]);
        }
        ans[queries[i].id] = res;
        if (queries[i].lc) {
            visit(tin[queries[i].lc]);
        }
    }
    for (int i = 1; i <= qid; i++) {
        printf("%d\n", ans[i]);
    }
}
```

1.9 Sparse Table

1.9.1 Sparse Table

```
int log_table[N], mx[LN][N], mn[LN][N];
inline void preprocess(){
    log_table[1] = 0;
    for(int i = 2; i <= n; i++) log_table[i] = log_table[i >> 1] + 1;

    for(int i = 1; i <= n; i++) mx[0][i] = a[i];
    for(int i = 1; i < LN; i++)
        for(int j = 1; j + (1 << i) - 1 <= n; j++)
            mx[i][j] = max(mx[i - 1][j] , mx[i - 1][j + (1 << (i - 1))]);

    for(int i = 1; i <= n; i++) mn[0][i] = b[i];
    for(int i = 1; i < LN; i++)
        for(int j = 1; j + (1 << i) - 1 <= n; j++)
            mn[i][j] = min(mn[i - 1][j] , mn[i - 1][j + (1 << (i - 1))]);
}

inline int get_max(int l, int r){
    int k = log_table[r - l + 1];
    return max(mx[k][l] , mx[k][r - (1 << k) + 1]);
}

inline int get_min(int l, int r){
    int k = log_table[r - l + 1];
    return min(mn[k][l] , mn[k][r - (1 << k) + 1]);
}
```

1.10 Treaps

1.10.1 Implicit Treap Cut And Paste

```
// Hackerrank Zalando Codesprint

#include "bits/stdc++.h"
using namespace std;

struct node{
    int left, right, pr, sz, val;
};

const int MAX = 100000 + 50;

node tree[MAX];
int N, root, null = 100000 + 5;

inline int create_node(int val){
    tree[N].pr  = rand();
    tree[N].sz  = 1;
    tree[N].val = val;
    tree[N].left = tree[N].right = null;
    return N++;
}

// Update Treap Information (when you're moving it around)
inline int upd(int x){
    int l = tree[x].left, r = tree[x].right;
    tree[x].sz = tree[l].sz + 1 + tree[r].sz;
    return x;
}

/*
   Takes the treap rooted at "rt" and puts the k smallest elements
   in it into sp.first, and the rest into sp.second
 */

inline pair < int, int > split(int rt, int k){
    if(rt >= null) return make_pair(null, null);
    pair < int, int > sp;
    if(tree[tree[rt].left].sz >= k){
        sp = split(tree[rt].left, k);
        tree[rt].left = sp.second;
        sp.second = upd(rt);
        return sp;
    }
    else{
        k -= tree[tree[rt].left].sz;
        sp = split(tree[rt].right, k - 1);
        tree[rt].right = sp.first;
        sp.first = upd(rt);
        return sp;
    }
}

// Merges treaps (l) and (r)
inline int merge(int l, int r){
    if(l >= null) return r;
    if(r >= null) return l;
    if(tree[l].pr > tree[r].pr){
        tree[l].right = merge(tree[l].right,r);
        return upd(l);
    }
    else{
        tree[r].left = merge(l,tree[r].left);
        return upd(r);
    }
}

/*
   Inserts element at position (i + 1) i.e after position (i) in the treap
 */

inline int insert(int i){
    pair < int, int > sp = split(root, i);
    int val; scanf("%d", &val);
    int x = create_node(val);
    sp.first = merge(sp.first,x);
    return merge(sp.first, sp.second);
}

/*
   Let arr[1...N] be the current array :-
   It looks like arr[1...i - 1] -> arr[i...j] -> arr[j + 1...N]
   work(i, j) takes subarray arr[i...j], cuts it, and pastes it at the beginning
   So the new array looks like arr[i..j] -> arr[1, i - 1] -> arr[j + 1..n]
 */

inline void work(int i, int j){
    pair < int, int > sp, sp2;
    sp = split(root, j);
    sp2 = split(sp.first, i - 1);
    root = merge(merge(sp2.second, sp2.first), sp.second);
}

/*
   Prints the in-order traversal of the treap i.e the sorted order of elements.
 */

inline void print(int idx){
    upd(idx);
    if(idx >= null) return;
    print(tree[idx].left);
    cout << tree[idx].val << ' ';
    print(tree[idx].right);
}

int main(){
    int n, q, i, j;
    scanf("%d", &n);
    root = null;
    while(n--) root = insert(N);
    scanf("%d", &q);
    while(q--){
        scanf("%d %d", &i, &j);
        work(i, j);
    }
    print(root);
}
```

1.10.2 Implicit Treap Reverse Subarray

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

const int NMAX = 40010;

struct node{
    int left, right, pr, sz, rev;
};
```

```
node tree[NMAX];
int N, null, root;

inline int create_node(){
    tree[N].pr = rand();
    tree[N].sz = 1;
    tree[N].left = tree[N].right = null;
    tree[N].rev = 0;
    return N++;
}

inline int upd(int x){
    int l = tree[x].left, r = tree[x].right;
    tree[x].sz = tree[l].sz + tree[r].sz + 1;
    return x;
}

// Swap left child and right child if it needs to be reversed.

inline void down(int rt){
    if(!tree[rt].rev) return;
    swap(tree[rt].left, tree[rt].right);
    tree[rt].rev = 0;
    tree[tree[rt].left].rev ^= 1, tree[tree[rt].right].rev ^= 1;
}


/*
   Takes the treap rooted at "rt" and puts the k smallest elements
   in it into sp.first, and the rest into sp.second
 */

inline pair < int, int > split(int rt, int k){
    if(rt >= null) return make_pair(null, null);
    down(rt);
    pair < int, int > sp;
    if(tree[tree[rt].left].sz >= k){
        sp = split(tree[rt].left, k);
        tree[rt].left = sp.second;
        sp.second = upd(rt);
        return sp;
    }
    else{
        k -= tree[tree[rt].left].sz;
        sp = split(tree[rt].right, k - 1);
        tree[rt].right = sp.first;
        sp.first = upd(rt);
        return sp;
    }
}

// Standard Treap Merge : down() is called to initiate reverse when needed.

inline int merge(int l, int r){
    if(l >= null) return r;
    if(r >= null) return l;
    if(tree[l].pr > tree[r].pr){
        down(l);
        tree[l].right = merge(tree[l].right, r);
        return upd(l);
    }
    else{
        down(r);
        tree[r].left = merge(l, tree[r].left);
        return upd(r);
    }
}


/*
   Returns the index (node no. in treap) of the (k + 1)th smallest value
   in the treap. In this problem, index equals value so printing the index
   suffices. However, if array values are different, then you should maintain
   a parameter 'val' in each treap node and print treap[idx].val.

   Note that this is an implicit treap, hence here we are simply returning
   the (k + 1)th value in the array, since the treap is ordered based on array
   indices.
 */

inline int search(int rt, int k){
    if(rt >= null) return rt;
    down(rt);
    if(tree[tree[rt].left].sz > k){
        return search(tree[rt].left, k);
    }
    else{
        k -= tree[tree[rt].left].sz;
        if(!k) return rt;
        return search(tree[rt].right, k - 1);
    }
}

/*
   Suppose array[1..N] is present. reverse(i, j) takes subarray [i...j] of it (1 based)
   and reverses it.
 */

inline void reverse(int i, int j){
    pair < int, int > sp, sp2;
    sp  = split(root, j); // sp.first = arr[1..j], sp.second = arr[j + 1...N]
    sp2 = split(sp.first, i - 1); // sp2.first = arr[1..i - 1], sp2.second = arr[i..j]
    tree[sp2.second].rev = 1; // sp2.second needs to be reversed, mark it.
    sp.first = merge(sp2.first,sp2.second); // Now merge everything normally!
    assert(merge(sp.first,sp.second) == root); // Merge
}

/*
   Insert element at position (i + 1) in the array i.e. after position (i)
   Here element value is not inputted since it's equal to index.
   Look at other codes for utilising this function
 */

inline int insert(int i){
    pair < int, int > sp = split(root, i);
    int x = create_node();
    sp.first = merge(sp.first, x);
    return merge(sp.first, sp.second);
}

int main(){
    int i, j, n;
    scanf("%d", &n);
    null = 40001;
    root = null;
    while(n--) root = insert(N);
    while(true){
        scanf("%d",&n);
        if(n >= 2) break;
        if(n){
            scanf("%d %d", &i, &j);
            reverse(i, j);
        }
        else{
            scanf("%d", &i);
            int ans = search(root, i - 1);
            printf("%d\n", ans + 1);
        }
    }
}
```

1.10.3 Normal Treap Fancy Queries

```
/*

   SPOJ TREAP

   I k : Insert k into S, if k is not in S

   D k : Delete k from S, if k is in S

   N i j : Print min{abs(S[x] - S[y]) | i <= x, y <= j} or -1 if the range has 1 element

   X i j : Print max{abs(S[x] - S[y]) | i <= x, y <= j} or -1 if the range has 1 element

 */

#include <bits/stdc++.h>
#define pii pair < int, int >
using namespace std;

const int MAXQ  = 2e5 + 5;
const int INF   = 1e9 + 9;
const int EMPTY = MAXQ - 1;

int N, Q, ROOT;
map < int, int > is_present;
char cmd[1];

struct treap_node{
    int val, pri, siz;
    int lc, rc;
    int maxv, minv, min_diff;
}treap[MAXQ];

inline int create_node(int val){
    N = N + 1;
    treap[N].val = val;
    treap[N].pri = rand();
    treap[N].siz = 1;
    treap[N].lc  = treap[N].rc = EMPTY;
    treap[N].maxv = treap[N].minv = val;
    treap[N].min_diff = INF;
    return N;
```

```
}

inline void refresh(int idx){
    treap[idx].siz = 1;
    treap[idx].maxv = treap[idx].minv = treap[idx].val;
    treap[idx].min_diff = INF;
    if(treap[idx].lc != EMPTY){
        treap[idx].siz += treap[treap[idx].lc].siz;
        treap[idx].maxv = max(treap[idx].maxv, treap[treap[idx].lc].maxv);
        treap[idx].minv = min(treap[idx].minv, treap[treap[idx].lc].minv);
        treap[idx].min_diff = min(treap[idx].min_diff, treap[treap[idx].lc].min_diff);
        treap[idx].min_diff = min(treap[idx].min_diff, treap[idx].val - treap[treap[idx].lc].maxv);
    }
    if(treap[idx].rc != EMPTY){
        treap[idx].siz += treap[treap[idx].rc].siz;
        treap[idx].maxv = max(treap[idx].maxv, treap[treap[idx].rc].maxv);
        treap[idx].minv = min(treap[idx].minv, treap[treap[idx].rc].minv);
        treap[idx].min_diff = min(treap[idx].min_diff, treap[treap[idx].rc].min_diff);
        treap[idx].min_diff = min(treap[idx].min_diff, treap[treap[idx].rc].minv - treap[idx].val);
    }
}

inline pii split(int idx, int key){
    pii parts = pii(EMPTY, EMPTY);
    if(idx == EMPTY) return parts;
    if(treap[idx].val <= key){
        parts = split(treap[idx].rc, key);
        treap[idx].rc = parts.first;
        parts.first = idx;
        refresh(idx);
    }
    else{
        parts = split(treap[idx].lc, key);
        treap[idx].lc = parts.second;
        parts.second = idx;
        refresh(idx);
    }
    return parts;
}

inline int merge(int l, int r){
    if(l == EMPTY) return r;
    if(r == EMPTY) return l;
    if(treap[l].pri > treap[r].pri){
        treap[l].rc = merge(treap[l].rc, r);
        refresh(l);
        return l;
    }
    else{
        treap[r].lc = merge(l, treap[r].lc);
        refresh(r);
        return r;
    }
}

inline int insert(int val){
    int idx = create_node(val);
    pii parts = split(ROOT, val - 1);
    return merge(merge(parts.first, idx), parts.second);
}

inline int erase(int val){
    pii y = split(ROOT, val);
    pii x = split(y.first, val - 1);
    return merge(x.first, y.second);
}

inline int get_kth(int idx, int k){
    if(treap[treap[idx].lc].siz >= k)
        return get_kth(treap[idx].lc, k);
    else{
        k -= treap[treap[idx].lc].siz;
        if(k == 1) return idx;
        return get_kth(treap[idx].rc, k - 1);
    }
}

inline int query(int val_i, int val_j){
    pii y = split(ROOT, val_j);
    pii x = split(y.first, val_i - 1);
    int res = treap[x.second].min_diff;
    ROOT = merge(merge(x.first, x.second), y.second);
    return res;
}

int main(){
    scanf("%d", &Q);
    ROOT = EMPTY;
    while(Q--){
        scanf("%s", cmd);
        int k, i, j;
        if(cmd[0] == 'I'){
            scanf("%d", &k);
            if(!is_present[k]){
                ROOT = insert(k);
                is_present[k] = 1;
            }
        }
        else if(cmd[0] == 'D'){
            scanf("%d", &k);
            if(is_present[k]){
                ROOT = erase(k);
                is_present[k] = 0;
            }
        }
        else if(cmd[0] == 'N'){
            scanf("%d %d", &i, &j);
            i++, j++;
            i = get_kth(ROOT, i), j = get_kth(ROOT, j);
            if(i == j) printf("-1\n");
            else printf("%d\n", query(treap[i].val, treap[j].val));
        }
        else{
            scanf("%d %d", &i, &j);
            i++, j++;
            i = get_kth(ROOT, i), j = get_kth(ROOT, j);
            if(i == j) printf("-1\n");
            else printf("%d\n", treap[j].val - treap[i].val);
        }
    }
}
```

1.10.4 Normal Treap Less Than K

```
/*
   SPOJ RaceTime
   1) Update A[i] = X for given i and X
   2) Print # of i such that L <= i <= R and A[i] <= X, for given L, R and X
 */

#include <bits/stdc++.h>
#define pii pair < int, int >
using namespace std;

const int MAXN  = 100005;
const int MAXQ  = 50005;
const int LN    = 20;
const int EMPTY = (MAXN + MAXQ) * LN - 1;

int N, n, q, arr[MAXN];
int treap_roots[MAXN];

struct treap_node{
    int val, pri, siz, lc, rc;
}treap[(MAXN + MAXQ) * LN];

inline int create_node(int val){
    N = N + 1;
    treap[N].val = val;
    treap[N].pri = rand();
    treap[N].siz = 1;
    treap[N].lc = treap[N].rc = EMPTY;
    return N;
}

inline void refresh(int root){
    treap[root].siz = treap[treap[root].lc].siz + 1 + treap[treap[root].rc].siz;
}

/*
   splits treap into two treaps parts.first and parts.second such that
   parts.first comprises all elements with val <= key and parts.second comprises
   all elements with val > key.
 */

inline pii split(int root, int key){
    pii parts = pii(EMPTY, EMPTY);
    if(root == EMPTY) return parts;
    if(treap[root].val <= key){
        parts = split(treap[root].rc, key);
        treap[root].rc = parts.first;
        refresh(root);
        parts.first = root;
        return parts;
    }
    else{
        parts = split(treap[root].lc, key);
        treap[root].lc = parts.second;
        refresh(root);
        parts.second = root;
        return parts;
    }
}
```

```
/*
   Merge treaps l, r.
   Note largest key in l must be <= smallest key in r
 */

inline int merge(int l, int r){
    if(l == EMPTY) return r;
    if(r == EMPTY) return l;
    if(treap[l].pri > treap[r].pri){
        treap[l].rc = merge(treap[l].rc, r);
        refresh(l);
        return l;
    }
    else{
        treap[r].lc = merge(l, treap[r].lc);
        refresh(r);
        return r;
    }
}

/*
   Insert treap_node named 'add' with value 'treap[add].val' into
   treap rooted at 'root'
 */

inline int insert(int root, int add){
    if(root == EMPTY) return add;
    pii parts = split(root, treap[add].val - 1);
    return merge(merge(parts.first, add), parts.second);
}

/*
   Remove 'rem_value' from treap rooted at 'root'
 */

inline int erase(int root, int rem_value){
    if(root == EMPTY) return EMPTY;
    if(treap[root].val == rem_value){
        return merge(treap[root].lc, treap[root].rc);
    }
    if(treap[root].val > rem_value){
        treap[root].lc = erase(treap[root].lc, rem_value);
        refresh(root);
        return root;
    }
    else{
        treap[root].rc = erase(treap[root].rc, rem_value);
        refresh(root);
        return root;
    }
}

/*
   Returns # of elements in the treap rooted at 'root' that
   has a value <= k.
 */

inline int query_k(int root, int k){
    if(root == EMPTY) return 0;
    if(treap[root].val <= k){
        return treap[treap[root].lc].siz + 1 + query_k(treap[root].rc, k);
```

```cpp
    }
    else{
        return query_k(treap[root].lc, k);
    }
}

/*
   Maintain a BIT in which each node is a TREAP.
   treap_roots[] denotes the roots of the treaps.
 */

inline void update(int idx, int val, int type){
    for(int i = idx; i <= n; i += i & -i){
        if(type) treap_roots[i] = insert(treap_roots[i], create_node(val));
        else     treap_roots[i] = erase(treap_roots[i], val);
    }
}

inline int query(int idx, int k){
    int res = 0;
    for(int i = idx; i > 0; i -= i & -i){
        res += query_k(treap_roots[i], k);
    }
    return res;
}

int main(){
    scanf("%d %d", &n, &q);
    for(int i = 1; i <= n; i++){
        treap_roots[i] = EMPTY;
    }
    for(int i = 1; i <= n; i++){
        scanf("%d", arr + i);
        update(i, arr[i], 1);
    }
    char buf[1];
    while(q--){
        scanf("%s", buf);
        if(buf[0] == 'M'){
            int i, x;
            scanf("%d %d", &i, &x);
            update(i, arr[i], 0);
            arr[i] = x;
            update(i, arr[i], 1);
        }
        else{
            int st, en, x;
            scanf("%d %d %d", &st, &en, &x);
            printf("%d\n", query(en, x) - query(st - 1, x));
        }
    }
}
```

1.10.5 Persistent Implicit Treap

```cpp
/*
   UVA Online Judge - Version Controlled IDE
 */

#include "bits/stdc++.h"
```

```
using namespace std;

const int NMAX = 10000000 + 50;
const int QMAX = 50000 + 50;
const int LMAX = 100 + 10;

struct node{
    int lft, rht, pr, sz, time;
    char ch;
};

int N, Q, cur_time;
int roots[QMAX];
char str[LMAX];
node T[NMAX];

const int null = (10000000) + 5;

inline int create_node(char c){
    assert(N < null);
    T[N].lft = T[N].rht = null;
    T[N].sz = 1;
    T[N].pr = rand();
    T[N].ch = c;
    T[N].time = cur_time;
    return N++;
}

/*
   For each insert and delete operation, create a new node instead of
   working with the original nodes (Persistence).
   This new node will have same data as the original. (except the time parameter)

   Since the expected height of the treap is log N, we will not create more than
   log N nodes for every split() operation.
 */

inline int copy(int x){ // Path Copying
    if(T[x].time == cur_time) return x;
    T[N] = T[x];
    T[N].time = cur_time;
    return N++;
}

// Update Treap Information
inline int upd(int x){
    T[x].sz = T[T[x].lft].sz + T[T[x].rht].sz + 1;
    return x;
}

/*
   Takes the treap rooted at "rt" and puts the k smallest elements
   in it into sp.first, and the rest into sp.second
 */

inline pair < int, int > split(int rt, int k, bool persistence){
    if(rt >= null) return make_pair(null,null);
    int l, x, r;
    pair < int, int > sp;
    // Do not destroy the original state if persistence is needed.
    if(persistence) rt = copy(rt); // Work with a copy of rt
    l = T[rt].lft, r = T[rt].rht;
    if(T[l].sz >= k){
        sp = split(l, k, persistence);
        T[rt].lft = sp.second;
        sp.second = upd(rt);
        return sp;
    }
    else{
        k -= T[l].sz;
        --k;
        sp = split(r, k, persistence);
        T[rt].rht = sp.first;
        sp.first = upd(rt);
        return sp;
    }
}

// Standard Treap Merge : Note that largest element of l <= smallest of r
inline int merge(int l, int r){
    if(l >= null) return r;
    if(r >= null) return l;
    if(T[l].pr > T[r].pr){
        T[l].rht = merge(T[l].rht, r);
        return upd(l);
    }
    else{
        T[r].lft = merge(l, T[r].lft);
        return upd(r);
    }
}

// Build Treap with n nodes, filled with characters in str[]
inline int build_treap(char *str, int n){
    int rt = null, i;
    for(i = 0; i < n; ++i)
        rt = merge(rt, create_node(str[i]));
    return rt;
}

// Insert required substring
inline int insert(int rt, int i, int n, char *str){
    int x = build_treap(str, n);
    pair < int, int > sp;
    sp = split(rt, i, 1);
    x = merge(sp.first, x);
    x = merge(x, sp.second);
    return x;
}

// Delete required substring
inline int remove(int rt, int i, int c){
    pair < int, int > sp1, sp2;
    sp1 = split(rt, i, 1);
    sp2 = split(sp1.second, c, 1);
    return merge(sp1.first, sp2.second);
}

// Prints in-order traversal of treap
inline int print_word(int rt){
    if(rt >= null) return 0;
    int ct = print_word(T[rt].lft);
    putchar(T[rt].ch);
    ct += (T[rt].ch == 'c');
    return ct + print_word(T[rt].rht);
}

// Print the required substring
inline int print(int rt, int i, int c){
    pair < int, int > sp1, sp2;
    sp1 = split(rt, i, 0);
    sp2 = split(sp1.second, c, 0);
    int ct = print_word(sp2.first);
    sp1.second = merge(sp2.first, sp2.second);
    merge(sp1.first, sp1.second);
    return ct;
}

int main(){
    scanf("%d", &Q);
    int i, t, c, ct, x;
    ct = 0;
    roots[cur_time++] = null;
    while(Q--){
        scanf("%d", &t);
        if(t == 1){
            scanf("%d %s", &i, str);
            i -= ct;
            roots[cur_time] = insert(roots[cur_time - 1], i, strlen(str), str);
            ++cur_time;
        }
        else if(t == 2){
            scanf("%d %d", &i, &c);
            i -= ct, c -= ct;
            --i;
            roots[cur_time] = remove(roots[cur_time - 1], i, c);
            ++cur_time;
        }
        else{
            scanf("%d %d %d", &x, &i, &c);
            x -= ct, i -= ct, c -= ct;
            --i;
            ct += print(roots[x], i, c);
            printf("\n");
        }
    }
}
```

1.11 Trie

1.11.1 Binary Trie

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

int n, k, x;

/*
   A subarray of a[] is beautiful if the bitwise xor of all the elements in the subarray
   is at least k. Print count of such subarrays.
 */

struct node{
    node *lc, *rc;
    int leaves;
    node(node *_lc = NULL, node *_rc = NULL, int _leaves = 0){
        lc = _lc;
        rc = _lc;
        leaves = _leaves;
    }
    inline int val(node *x){
        return x ? x -> leaves : 0;
    }
    inline void create(node* &x){
        if(!x) x = new node();
    }
    inline int query(int pos, int prefix){
        if(pos == -1) return 0;
        int k_bit = k & (1 << pos);
        int p_bit = prefix & (1 << pos);
        int res = 0;
        if(!k_bit){
            if(!p_bit){
                res += val(rc);
                create(lc);
                res += lc -> query(pos - 1, prefix);
            }
            else{
                res += val(lc);
                create(rc);
                res += rc -> query(pos - 1, prefix);
            }
        }
        else{
            if(p_bit){
                create(lc);
                res += lc -> query(pos - 1, prefix);
            }
            else{
                create(rc);
                res += rc -> query(pos - 1, prefix);
            }
        }
        return res;
    }
    node *insert(int pos, long long prefix){
        if(pos == -1){
            ++leaves;
            return this;
        }
        ++leaves;
        int p_bit = prefix & (1 << pos);
        if(!p_bit){
            create(lc);
            lc = lc -> insert(pos - 1, prefix);
        }
        else{
            create(rc);
            rc = rc -> insert(pos - 1, prefix);
        }
        return this;
    }
};

node *trie = new node();
```

```
int main(){
    freopen("ioi.in", "r", stdin);
    scanf("%d %d", &n, &k);
    k--;
    long long res  = 0;
    int prefix_xor = 0;
    for(int i = 0; i < n; i++){
        scanf("%d", &x);
        prefix_xor ^= x;
        res += trie ->  query(30, prefix_xor) + (prefix_xor > k);
        trie = trie -> insert(30, prefix_xor);
    }
    printf("%lld\n", res);
}
```

1.11.2 Persistent Trie

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

const int N = 300000 + 30;
const int BIT = 30;

struct trie_node {
    trie_node *lc, *rc;
    trie_node(trie_node *l = NULL, trie_node *r = NULL) {
        lc = l;
        rc = r;
    }
    inline trie_node* get_left(trie_node* x) {
        if (x) {
            return x -> lc;
        } else {
            return NULL;
        }
    }
    inline trie_node* get_right(trie_node* x) {
        if (x) {
            return x -> rc;
        } else {
            return NULL;
        }
    }
    inline trie_node* update(trie_node* old_root, int i, int value) {
        if (i == -1) {
            return new trie_node();
        }
        trie_node * nw = new trie_node();
        if (value & (1 << i)) {
            nw -> lc = get_left(old_root);
            nw -> rc = update(get_right(old_root), i - 1, value);
        } else {
            nw -> rc = get_right(old_root);
            nw -> lc = update(get_left(old_root), i - 1, value);
        }
        return nw;
    }
    inline int query_min(trie_node* root, int i, int value) {
        if (i == -1) {
            return 0;
        }
        if (value & (1 << i)) {
            if (get_right(root)) {
                return query_min(get_right(root), i - 1, value);
            } else {
                return query_min(get_left(root), i - 1, value) | (1 << i);
            }
        } else {
            if (get_left(root)) {
                return query_min(get_left(root), i - 1, value);
            } else {
                return query_min(get_right(root), i - 1, value) | (1 << i);
            }
        }
    }
    inline int query_max(trie_node* root, int i, int value) {
        if (i == -1) {
            return 0;
        }
        if (value & (1 << i)) {
            if (get_left(root)) {
                return query_max(get_left(root), i - 1, value) | (1 << i);
            } else {
                return query_max(get_right(root), i - 1, value);
            }
        } else {
            if (get_right(root)) {
                return query_max(get_right(root), i - 1, value) | (1 << i);
            } else {
                return query_max(get_left(root), i - 1, value);
            }
        }
    }
};

int parent[N];
unordered_map < int, int > compress;
trie_node* trie[N];

int main() {
    int n, q, r, root_key;
    scanf("%d %d %d %d", &n, &q, &r, &root_key);
    int timer = 0;
    trie[0] = new trie_node();
    compress[r] = ++timer;
    parent[compress[r]] = 0;
    trie[compress[r]] = trie[0] -> update(trie[parent[compress[r]]], BIT, root_key);
    for (int i = 1; i < n; i++) {
        int u, v, k;
        scanf("%d %d %d", &u, &v, &k);
        compress[u] = ++timer;
        parent[compress[u]] = compress[v];
        trie[compress[u]] = trie[0] -> update(trie[parent[compress[u]]], BIT, k);
    }
    int last_answer = 0;
    while (q--) {
        int t; scanf("%d", &t);
        t ^= last_answer;
        if (t == 0) {
            int v, u, k; scanf("%d %d %d", &v, &u, &k);
            v ^= last_answer; u ^= last_answer; k ^= last_answer;
            compress[u] = ++timer;
            parent[compress[u]] = compress[v];
            trie[compress[u]] = trie[0] -> update(trie[parent[compress[u]]], BIT, k);
        } else {
            int v, k; scanf("%d %d", &v, &k);
            v ^= last_answer; k ^= last_answer;
            int min_answer = trie[0] -> query_min(trie[compress[v]], BIT, k);
            int max_answer = trie[0] -> query_max(trie[compress[v]], BIT, k);
            printf("%d %d\n", min_answer, max_answer);
            last_answer = min_answer ^ max_answer;
        }
    }
}
```

1.12 Union Of Rectangles

1.12.1 Area Of Union Of Rectangles

```
// Problem: Strange Tree (OpenBracket 2017).

#include "bits/stdc++.h"
using namespace std;

// Area of Union of Rectangles Template

// Given a list of rectangles on the 2D Plane, returns the union of their areas.
// A rectangle is defined as a pair of pairs ((x1, y1), (x2, y2)) where (x1, y1)
// is the bottom left corner of the rectangle and (x2, y2) is the top right corner.
// O(n log n)

const int MAXQ = 400005; // 2 * number of rectangles.
const int MINY = 0;
const int MAXY = 400005; // The max y coordinate.
#define ii pair < int, int >

struct event{
    int x, l, r, t;
    event(int _x = 0, int _l = 0, int _r = 0, int  _t = 0){
        x = _x, l = _l, r = _r, t = _t;
    }
    friend bool operator < (event a, event b){
        if(a.x == b.x){
            int y1 = (a.t == 1) ? (a.l) : (a.r);
            int y2 = (b.t == 1) ? (b.l) : (b.r);
            return (y1 < y2);
        }
        return (a.x < b.x);
    }
}query[MAXQ];

int lazy[MAXY * 4];
ii tree[MAXY * 4];

inline ii combine(ii a, ii b){
    if(a.first == b.first) return ii(a.first, a.second + b.second);
    return min(a, b);
}

inline void push(int l, int r, int node){
    tree[node].first += lazy[node];
    if(l != r){
        lazy[node * 2] += lazy[node];
        lazy[node * 2 + 1] += lazy[node];
    }
    lazy[node] = 0;
}

inline void update(int node, int l, int r, int qs, int qe, int val){
    push(l, r, node);
    if(l > qe || r < qs) return;
    if(l >= qs && r <= qe){
        lazy[node] = val;
        push(l, r, node);
        return;
    }
    int mid = (l + r) >> 1;
    update(node * 2, l, mid, qs, qe, val);
    update(node * 2 + 1, mid + 1, r, qs, qe, val);
    tree[node] = combine(tree[node * 2], tree[node * 2 + 1]);
}

inline void build(int node, int l, int r){
    tree[node] = ii(0, r - l + 1);
    if(l != r){
        int mid = (l + r) >> 1;
        build(node * 2, l, mid);
        build(node * 2 + 1, mid + 1, r);
    }
}

inline long long solve(vector < pair < ii, ii > > rectangles) {
    int counter = 0;
    for(pair < ii, ii > r: rectangles){
        ++counter;
        int x1 = r.first.first;
        int y1 = r.first.second;
        int x2 = r.second.first;
        int y2 = r.second.second;
        query[2 * counter - 1] = event(x1, y1, y2, 1);
        query[2 * counter] = event(x2, y1, y2, -1);
    }
    sort(query + 1, query + 2 * counter + 1);
    build(1, MINY, MAXY);
    long long area = 0;
    for(int i = 1; i <= 2 * counter; i++){
        update(1, MINY, MAXY, query[i].l, query[i].r - 1, query[i].t);
        long long min_val = tree[1].first;
        long long min_cnt = (!min_val) ? (tree[1].second) : (0);
        area += (query[i + 1].x - query[i].x) * 1LL * (MAXY - MINY + 1 - min_cnt);
    }
    return area;
}

// End of template

const int N = 200005;
const int LN = 20;

int n, timer;
int arr[N], tin[N], tout[N], depth[N];
int anc[LN][N];
vector < int > adj[N];
vector < int > values[N];
```

```
inline void dfs(int u, int p) {
    values[arr[u]].push_back(u);
    anc[0][u] = p;
    for (int i = 1; i < LN; i++)
        anc[i][u] = anc[i - 1][anc[i - 1][u]];
    tin[u] = ++timer;
    for (int v: adj[u]) {
        if (v != p) {
            depth[v] = depth[u] + 1;
            dfs(v, u);
        }
    }
    tout[u] = timer;
}

inline bool is_ancestor(int x, int y) {
    return tin[x] <= tin[y] and tout[x] >= tout[y];
}

inline int get_ancestor(int x, int k) {
    for (int i = LN - 1; i >= 0; i--) {
        if (k >= (1 << i)) {
            x = anc[i][x];
            k -= (1 << i);
        }
    }
    return x;
}

int main() {
    ios :: sync_with_stdio(false);
    cin >> n;
    for (int i = 2; i <= n; i++) {
        int p; cin >> p;
        adj[p].push_back(i);
        adj[i].push_back(p);
    }
    for (int i = 1; i <= n; i++) {
        cin >> arr[i];
    }
    dfs(1, 1);
    vector < pair < ii, ii > > rectangles;
    int x1, y1, x2, y2;
    for (int i = 1; i <= n / 2; i++) {
        int x = values[i][0];
        int y = values[i][1];
        if (tin[x] > tin[y]) swap(x, y);
        if (is_ancestor(x, y)) {
            int p = get_ancestor(y, depth[y] - depth[x] - 1);
            x1 = 1, x2 = tin[p];
            y1 = tin[y], y2 = tout[y] + 1;
            rectangles.push_back(make_pair(ii(x1, y1), ii(x2, y2)));
            x1 = tin[y], x2 = tout[y] + 1;
            y1 = tout[p] + 1, y2 = n + 1;
            rectangles.push_back(make_pair(ii(x1, y1), ii(x2, y2)));
        } else {
            x1 = tin[x], x2 = tout[x] + 1;
            y1 = tin[y], y2 = tout[y] + 1;
            rectangles.push_back(make_pair(ii(x1, y1), ii(x2, y2)));
        }
    }
    long long bad_pairs = solve(rectangles);
    cout << (n * 1LL * n - bad_pairs * 2LL) << endl;
}
```

1.13 Wavelet Tree

1.13.1 Wavelet Tree

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

typedef vector < int > :: iterator iter;


/*
   ------------ Wavelet Tree Template ----------

   quantile(k, a, b) : k'th smallest element in [a, b)
   range(x, y, a, b) : # of elements with value in range [x, y] in subarray [a, b)
   rank(x, k) : # of occurrences of x in [0, k)
   push_back(x) : Append another value x to the existing array.
Note : x should be in [0, sigma)
pop_back() :   Pop the last element from the existing array.
swap_adj(i) :  Swap arr[i] and arr[i + 1]. Assumes i is in [0, n - 1)

WaveTree obj(arr, sigma) : Creates a Wavelet Tree on the vector 'arr', alphabet size [0, sigma)

All indices are Zero-Based.

-----------------------------------------------
 */


class WaveTree {
    vector < vector < int > > tree;
    vector < int > arr_copy;
    // tree[u][i] = uptil index (i) in node (u), how many values are <= (mid)
    int n, s;
    // O(n * log (sigma)) construction
    inline void build(iter b, iter e, int l, int r, int u) {
        if (l == r) return;
        int m = (l + r) / 2;
        tree[u].reserve(e - b + 1);
        tree[u].push_back(0);
        for (iter it = b; it != e; ++it)
            tree[u].push_back(tree[u].back() + (*it <= m));
        iter p = stable_partition(b, e, [=](int i){ return i <= m;});
        // arr[b, p) have elements <= m and arr[p, e) have > m
        build(b, p, l, m, u * 2);
        build(p, e, m + 1, r, u * 2 + 1);
    }

    int qq, w;
    inline int range(int a, int b, int l, int r, int u) {
        if (r < qq or w < l) return 0;
        if (qq <= l and r <= w) return b - a;
        int m = (l + r) / 2, za = tree[u][a], zb = tree[u][b];
        return range(za, zb, l, m, u * 2) +
            range(a - za, b - zb, m + 1, r, u * 2 + 1);
    }
```

```
public:

//arr[i] in [0, sigma)
WaveTree(vector < int > arr, int sigma) {
    n = arr.size();
    s = sigma;
    tree.resize(s * 2);
    arr_copy = arr;
    build(arr.begin(), arr.end(), 0, s - 1, 1);
}

//k in [1, n], [a, b) is 0-indexed, -1 if error
inline int quantile(int k, int a, int b) {
    if (a < 0 or b > n or k < 1 or k > b - a) return -1;
    int l = 0, r = s - 1, u = 1, m, za, zb;
    while (l != r) {
        m = (l + r) / 2;
        za = tree[u][a];
        zb = tree[u][b];
        u *= 2;
        if (k <= zb - za)
            a = za, b = zb, r = m;
        else
            k -= zb - za, a -= za, b -= zb,
              l = m + 1, ++u;
    }
    return r;
}

//Counts numbers in [x, y] in positions [a, b)
inline int range(int x, int y, int a, int b) {
    if (y < x or b <= a) return 0;
    qq = x; w = y;
    return range(a, b, 0, s - 1, 1);
}

//Count occurrences of x in positions [0, k)
inline int rank(int x, int k) {
    int l = 0, r = s - 1, u = 1, m, z;
    while (l != r) {
        m = (l + r) / 2;
        z = tree[u][k];
        u *= 2;
        if(x <= m) k = z, r = m;
        else k -= z, l = m + 1, ++u;
    }
    return k;
}

//x in [0, sigma)
inline void push_back(int x) {
    int l = 0, r = s - 1, u = 1, m, p;
    ++n;
    while (l != r) {
        m = (l + r)/2;
        p = (x <= m);
        tree[u].push_back(tree[u].back() + p);
        u *= 2;
        if(p) r = m;
        else l = m + 1, ++u;
    }
    }

    //Assumes that array is non-empty
    inline void pop_back() {
        int l = 0, r = s - 1, u = 1, m, p, k;
        --n;
        while (l != r) {
            m = (l + r) / 2;
            k = tree[u].size();
            p = tree[u][k - 1] - tree[u][k - 2];
            tree[u].pop_back();
            u *= 2;
            if(p) r = m;
            else l = m + 1, ++u;
        }
    }

    //swap arr[i] with arr[i + 1], i in [0, n - 1)
    inline void swap_adj(int i){
        int &x = arr_copy[i], &y = arr_copy[i + 1];
        int l = 0, r = s - 1, u = 1;
        while(l != r){
            int m = (l + r) / 2, p = (x <= m), q = (y <= m);
            if (p != q){
                tree[u][i + 1] ^= tree[u][i] ^ tree[u][i + 2];
                break;
            }
            int z = tree[u][i];
            u *= 2;
            if(p) i = z, r = m;
            else i -= z, l = m + 1, ++u;
        }
        swap(x, y);
    }
};

int main() {

    int n, q;
    scanf("%d %d", &n, &q);
    vector < int > arr(n);
    for(int i = 0; i < n; i++) scanf("%d", &arr[i]);

    //Co-ordinate Compression
    vector < int > values;
    for(int i = 0; i < n; i++){
        values.push_back(arr[i]);
    }

    sort(values.begin(), values.end());
    values.resize(unique(values.begin(), values.end()) - values.begin());

    int sigma = 0;
    vector < int > orig(n);
    for(int i = 0; i < n; i++){
        int init = arr[i];
        arr[i] = lower_bound(values.begin(), values.end(), arr[i]) - values.begin();
        orig[arr[i]] = init;
        sigma = max(sigma, arr[i]);
    }

    /*
```

```
       1) Vector 'arr' represents the array
       2) 'sigma' represents the alphabet size i.e [0, sigma + 1) in this case.
     */

    WaveTree wt(arr, sigma + 1);

    for(int qq = 0; qq < q; qq++){
        int cmd, i, k;
        scanf("%d", &cmd);
        if(cmd){
            scanf("%d", &i);
            wt.swap_adj(i);
        }
        else{
            scanf("%d %d", &i, &k);
            // val = 'k'th smallest element in [0, i + 1)
            int val = orig[wt.quantile(k, 0, i + 1)];
            printf("%d\n", val);
        }
    }
}
```

2 Flows and 2SAT

2.1 2SAT

2.1.1 2SAT Template

```
// CF - The Door Problem
#include "bits/stdc++.h"
using namespace std;

const int N = 100000;

/*
   2-SAT Template
   Given an implication graph, this checks if a solution exists.

   addXor(), addAnd(), addOr() can be used to appropriately add clauses.
   forceTrue() forces some variable to be true.
   forceFalse() forces some variable to be false.

   You can also add additional implications yourself.
   init() initializes 2-SAT arrays.
   solve() checks if in the final implication graph, a valid solution exists.
   mark[u] stores the boolean value of the node (u). You can use mark[] to
   recover the final solution as well.

   Notes on Indexing Nodes :
   u = 2k, !u = 2k + 1
   Nodes are 0-indexed. [0, NUM_VERTICES)
 */

int NUM_VERTICES, id;
int arr[N * 2];
vector < int > adj[N * 2];
bool mark[N * 2];

inline bool dfs(int node) {
    if (mark[node ^ 1]) {
        return false;
    }
    if (mark[node]) {
        return true;
    }
    mark[node] = true;
    arr[id++] = node;
    for (int i = 0; i < (int) adj[node].size(); i++) {
        if (!dfs(adj[node][i])) {
            return false;
        }
    }
    return true;
}

inline void init() {
    for (int i = 0; i < NUM_VERTICES; i++) {
        adj[i].clear();
    }
    memset(mark, 0, sizeof(mark));
}

// Adds the clause (u or v) to the set of clauses
inline void addOr(int u, int v) {
    adj[u ^ 1].push_back(v);
    adj[v ^ 1].push_back(u);
}

// Adds the clause (u == v) to the set of clauses
inline void addEquivalent(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
    adj[u ^ 1].push_back(v ^ 1);
    adj[v ^ 1].push_back(u ^ 1);
}

// Adds the clause (u xor v) to the set of clauses
inline void addXor(int u, int v) {
    addOr(u, v);
    addOr(u ^ 1, v ^ 1);
}

// Forces variable (u) to be true
inline void forceTrue(int u) {
    adj[u ^ 1].push_back(u);
}

// Forces variable (u) to be false
inline void forceFalse(int u) {
    adj[u].push_back(u ^ 1);
}

// Adds the clause (u and v) to the set of clauses
inline void addAnd(int u, int v) {
    forceTrue(u);
    forceTrue(v);
}

// Returns true if a solution exists.
inline bool solve() {
    for (int i = 0; i < NUM_VERTICES; i++) {
        sort(adj[i].begin(), adj[i].end());
        adj[i].resize(unique(adj[i].begin(), adj[i].end()) - adj[i].begin());
    }
    for (int i = 0; i < NUM_VERTICES; i += 2) {
        if ((!mark[i]) && (!mark[i + 1])) {
            id = 0;
            if(!dfs(i)) {
                while (id > 0) {
                    mark[arr[--id]] = false;
                }
                if(!dfs(i + 1)) {
                    return false;
                }
            }
        }
    }
    return true;
}

// End of 2-SAT Template.

int n, m;
int r[N];
vector < int > switches[N];

int main() {
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        cin >> r[i];
    }
    for (int i = 0; i < m; i++) {
        int x; cin >> x;
        while (x--) {
            int foo; cin >> foo;
            switches[foo - 1].push_back(i);
        }
    }
    NUM_VERTICES = 2 * m; init();
    for (int i = 0; i < n; i++) {
        int x = switches[i][0], y = switches[i][1];
        if (r[i] == 1) {
            addEquivalent(x * 2, y * 2);
        } else {
            addXor(x * 2, y * 2);
        }
    }
    bool ok = solve();
    if (ok) {
        cout << "YES\n";
    } else {
        cout << "NO\n";
    }
}
```

2.2 Bipartite Matching

2.2.1 BPM Template

```
// 2014 ICPC World Finals: Problem I
```

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

/*
   Hopcroft Karp Max Matching in O(E * sqrt(V))
   MAXN = Number of Nodes, MAXM = Number of Edges
   n1 = Size of left partite, n2 = Size of right partite
   Nodes are numbered from [0, n1 - 1] and [0, n2 - 1]

   init(n1, n2) declares the two partite sizes and resets arrays
   addEdge(x, y) adds an edge between x in left partite and y in right partite
   maxMatching() returns the maximum matching

   For the two functions below: L denotes the list of nodes in the left partite,
   R denotes the list of nodes in the right partite, and edges denotes the list
   of edges between a node in L and a node in R. Make sure that the graph is bipartite.

   constructMVC(L, R, edges) returns the set of nodes in the minimum vertex cover
   constructMIS(L, R, edges) returns the set of nodes in the maximal independent set

   Important Theorems:

   |Maximum Matching| = |Minimum Vertex Cover| (Konig's Theorem)
   |L| + |R| - Maximum Matching = Maximal Independent Set
 */

const int MAXN = 105;
const int MAXM = 10005;

set < int > adjList[MAXN];
int seen[MAXN];
int matched[MAXN];
int n1, n2, edges, last[MAXN], previous[MAXM], head[MAXM];
int matching[MAXN], dist[MAXN], Q[MAXN];
bool used[MAXN], vis[MAXN];

inline void init(int _n1, int _n2) {
    n1 = _n1;
    n2 = _n2;
    edges = 0;
    fill(last, last + n1, -1);
    fill(matching, matching + n2, -1);
}

inline void addEdge(int u, int v) {
    head[edges] = v;
    previous[edges] = last[u];
    last[u] = edges++;
}

inline void bfs() {
    fill(dist, dist + n1, -1);
    int sizeQ = 0;
    for(int u = 0; u < n1; ++u){
        if(!used[u]){
            Q[sizeQ++] = u;
            dist[u] = 0;
        }
    }
    for(int i = 0; i < sizeQ; i++){
        int u1 = Q[i];
        for(int e = last[u1]; e >= 0; e = previous[e]){
            int u2 = matching[head[e]];
            if(u2 >= 0 && dist[u2] < 0){
                dist[u2] = dist[u1] + 1;
                Q[sizeQ++] = u2;
            }
        }
    }
}

inline bool dfs(int u1) {
    vis[u1] = true;
    for(int e = last[u1]; e >= 0; e = previous[e]){
        int v = head[e];
        int u2 = matching[v];
        if((u2 < 0) || (!vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2))){
            matching[v] = u1;
            used[u1] = true;
            return true;
        }
    }
    return false;
}

inline int maxMatching() {
    fill(used, used + n1, false);
    fill(matching, matching + n2, -1);
    for(int res = 0; ;){
        bfs();
        fill(vis, vis + n1, false);
        int f = 0;
        for(int u = 0; u < n1; ++u)
            if(!used[u] && dfs(u))
                ++f;
        if(!f) return res;
        res += f;
    }
}

inline void alternating_paths(int u, int c = 0) {
    seen[u] = true;
    for (int v: adjList[u]) {
        if (seen[v]) continue;
        if (c == 0) {
            if (matching[v] == -1) continue;
            alternating_paths(v, c ^ 1);
        } else if (v == matching[u]) {
            alternating_paths(v, c ^ 1);
        }
    }
}

inline void initGraph(vector < pair < int, int > > edges) {
    for (int i = 0; i < MAXN; i++) {
        adjList[i].clear();
    }
    for (int i = 0; i < (int) edges.size(); i++) {
        adjList[edges[i].first].insert(edges[i].second);
        adjList[edges[i].second].insert(edges[i].first);
    }
}

// Constructs the Minimum Vertex Cover using entries in matching[]
// Assumes maxMatching() has been called just before this function is called.
// Considers ONLY nodes in L, R (left partite and right partite)
// edges: List of edges between nodes in L and nodes in R in this Bipartite Graph.
set < int > constructMVC(vector < int > L, vector < int > R, vector < pair < int, int > > edges) {
    initGraph(edges);
    fill(matched, matched + MAXN, false);
    fill(seen, seen + MAXN, false);
    for (int r: R) {
        if (matching[r] != -1) {
            matched[matching[r]] = true;
        }
    }
    for (int l: L) {
        if (matched[l] == true) continue;
        if (!seen[l]) {
            alternating_paths(l);
        }
    }
    set < int > mvc;
    for (int r: R) {
        if (seen[r]) {
            mvc.insert(r);
        }
    }
    for (int l: L) {
        if (!seen[l]) {
            mvc.insert(l);
        }
    }
    return mvc;
}

// Constructs the Maximal Independent Set using entries in matching[]
// Assumes maxMatching() has been called just before this function is called!
// Considers ONLY nodes in L, R (left partite and right partite)
// edges: List of edges between nodes in L and nodes in R in this Bipartite Graph.
set < int > constructMIS(vector < int > L, vector < int > R, vector < pair < int, int > > edges) {
    set < int > U;
    for (int vertex: L) U.insert(vertex);
    for (int vertex: R) U.insert(vertex);
    set < int > MVC = constructMVC(L, R, edges);
    for (int vertex: MVC) U.erase(vertex);
    return U;
}

/******** End of Bipartite Matching Template ********/

const int N = 105;
const int M = N * N;

int n, d;
set < int > adj[N];
int x[N], y[N], visited[N];
vector < int > black, white;

inline int euclidean_distance(int i, int j) {
    return (x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j]);
}

inline void coloring(int u, int color) {
    visited[u] = true;
    if (!color) {
        black.push_back(u);
    } else {
        white.push_back(u);
    }
    for (int v: adj[u]) {
        if (!visited[v]) {
            coloring(v, color ^ 1);
        }
    }
}

int solve(int s, int t, int threshold) {
    set < int > candidates;
    for (int i = 0; i < n; i++) {
        adj[i].clear();
        visited[i] = false;
        if (s == i || t == i) {
            continue;
        }
        if (euclidean_distance(s, i) <= threshold && euclidean_distance(t, i) <= threshold) {
            candidates.insert(i);
        }
    }
    for (int i: candidates) {
        for (int j: candidates) {
            if (i != j) {
                if (euclidean_distance(i, j) > threshold) {
                    adj[i].insert(j);
                    adj[j].insert(i);
                }
            }
        }
    }
    black.clear();
    white.clear();
    for (int i = 0; i < n; i++) {
        if (!candidates.count(i)) {
            continue;
        }
        if (!visited[i]) {
            coloring(i, 0);
        }
    }
    init(n, n);
    for (int w: white) {
        for (int b: black) {
            if (adj[w].count(b))
                addEdge(w, b);
        }
    }
    int clique_size = 2 + (int) white.size() + (int) black.size() - maxMatching();
    return clique_size;
}

int main() {
    ios :: sync_with_stdio(false);
    cin >> n >> d;
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }
    int result = 1;
    set < int > sol;
    sol.insert(0);
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int edge_dist = euclidean_distance(i, j);
            if (edge_dist <= d * d) {
                int best = solve(i, j, euclidean_distance(i, j));
                if (best > result) {
                    result = best;
                    sol.clear(); sol.insert(i); sol.insert(j);
                    vector < pair < int, int > > edges;
                    for (int u = 0; u < n; u++) {
                        for (int v: adj[u]) {
                            edges.push_back(make_pair(min(u, v), max(u, v)));
                        }
                    }
                    set < int > mis = constructMIS(white, black, edges);
                    for (int vertex: mis) {
                        sol.insert(vertex);
                    }
                }
            }
        }
    }
    cout << (int) sol.size() << endl;
    for (int v: sol) {
        cout << (v + 1) << ' ';
    }
    cout << endl;
}
```

2.2.2 Semi Dynamic Matching

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

const int MAX_N = 56789;
const int MAX_S = 12;
const int MAX_V = 1234567;

int t, n, ans, timer;
int dice[MAX_N][MAX_S];
int match[MAX_N];
int vis[MAX_V];
vector < int > values[MAX_V];

void cleanup() {
    ans = 0;
    timer = 0;
    for (int i = 0; i < MAX_V; i++) {
        values[i].clear();
        vis[i] = 0;
    }
}

void init() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= 6; j++) {
            cin >> dice[i][j];
            values[dice[i][j]].push_back(i);
        }
    }
}

/**
  For visualization purposes, let's assume the left partite are the numbers
  and the right partite are the dice indices. We want to check if there exists
  a perfect matching, i.e. all the numbers are covered.

  Let's maintain an array match[] where match[x] denotes the number that dice x has
  been matched to. We already have values[] computed in the above init() function
  which tells us which dice have a particular number.

  We use the recursive bipartite matching algorithm to check for existence.
 **/

bool dfs(int x) {
    // Can we augment x into our existing matching?
    vis[x] = timer;
    for (int dice_id: values[x]) {
        if (match[dice_id] == -1) {
            // We found a dice which has not been matched to anyone, yay!
            match[dice_id] = x;
            return true;
        }
    }
    // Ok, so there isn't any "free" dice.
    // We can fix a dice, recurse and see if we can get a solution.
    for (int dice_id: values[x]) {
        int num = match[dice_id];
        if (vis[num] != timer && dfs(num)) {
            match[dice_id] = x;
            return true;
        }
    }
    return false;
}

bool check(int l, int r) {
    if (r - l + 1 > n) {
        return false;
    }
    // We have some perfect matching (possibly empty).
    // We want to add the number (r) to it and see if it still holds.

    // Mark all numbers in this range as unvisited.
    ++timer;

    // We'll try to augment our matching by adding (r) to it.
    // This follows the recursive bipartite matching algorithm.
    return dfs(r);
}

void slide(int l) {
    // We cannot get any more straights with (l), so lets slide our window
    // Basically mark all dice associated with (l) as free.
    for (int dice_id: values[l]) {
        if (match[dice_id] == l) {
            match[dice_id] = -1;
        }
    }
}
```

```
void solve() {
    for (int i = 1; i <= n; i++) {
        match[i] = -1; // All dice are unmatched.
    }
    int l = 1, r = 1;
    while (r < MAX_V) {
        if (check(l, r)) {
            ans = max(ans, r - l + 1);
            r += 1;
        } else {
            slide(l);
            l += 1;
            r = max(l, r);
        }
    }
}

int main() {
    freopen("inp.in", "r", stdin);
    freopen("A_large.out", "w", stdout);
    ios :: sync_with_stdio(false);
    cin >> t;
    for (int qq = 1; qq <= t; qq++) {
        cleanup();
        init();
        solve();
        printf("Case #%d: %d\n", qq, ans);
    }
}
```

2.3 Directed Graphs (Matching)

2.3.1 Maximum Antichain Dilworths

```
// LIGHTOJ
#include "bits/stdc++.h"
using namespace std;

const int N = 105;
const int M = 105 * 105;

int t, n;
int arr[N];
vector < int > values;

/*
   We will use Dilworths' Theorem and Min-Path-Cover on a DAG to solve this problem.
   Add an edge from number x to number y (x != y), if y % x == 0.
   Now we've built a dag, and we want to find the size of maximum antichain in this DAG.
   We need to find a subset of nodes such that no node in the subset can be reached from
   any other node in the subset.

   By Dilworth's Theorem, Size of maximum antichain = Min Path Cover in the DAG.
   We also know that Min Path Cover is just given by n - maxMatching(). Done!

   Computing the Lexicographically Smallest Anti-Chain:
   Let the size be S.
   To find the lexicographically smallest anti-chain, fix the smallest element and find
   the maximum antichain on the remaining graph. If you can get an antichain of size S - 1,
   then the smallest element can be taken. Repeat this process!
 */

/*
   Standard Matching Template
 */

int n1, n2, edges, last[N], previous[M], head[M];
int matching[N], dist[N], Q[N];
bool used[N], vis[N];

inline void init(int _n1, int _n2) {
    n1 = _n1;
    n2 = _n2;
    edges = 0;
    fill(last, last + n1, -1);
}

inline void addEdge(int u, int v) {
    head[edges] = v;
    previous[edges] = last[u];
    last[u] = edges++;
}

inline void bfs() {
    fill(dist, dist + n1, -1);
    int sizeQ = 0;
    for(int u = 0; u < n1; ++u){
        if(!used[u]){
            Q[sizeQ++] = u;
            dist[u] = 0;
        }
    }
    for(int i = 0; i < sizeQ; i++){
        int u1 = Q[i];
        for(int e = last[u1]; e >= 0; e = previous[e]){
            int u2 = matching[head[e]];
            if(u2 >= 0 && dist[u2] < 0){
                dist[u2] = dist[u1] + 1;
                Q[sizeQ++] = u2;
            }
        }
    }
}

inline bool dfs(int u1) {
    vis[u1] = true;
    for(int e = last[u1]; e >= 0; e = previous[e]){
        int v = head[e];
        int u2 = matching[v];
        if(u2 < 0 || !vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2)){
            matching[v] = u1;
            used[u1] = true;
            return true;
        }
    }
    return false;
}

inline int maxMatching() {
    fill(used, used + n1, false);
    fill(matching, matching + n2, -1);
    for(int res = 0; ;){
        bfs();
        fill(vis, vis + n1, false);
        int f = 0;
        for(int u = 0; u < n1; ++u)
            if(!used[u] && dfs(u))
                ++f;
        if(!f) return res;
        res += f;
    }
}

// End of Hopcroft Karp Template


int main(){
    scanf("%d", &t);
    for(int qq = 1; qq <= t; qq++){
        scanf("%d", &n);
        values.clear();
        for(int i = 0; i < n; i++){
            scanf("%d", arr + i);
            values.push_back(arr[i]);
        }
        sort(values.begin(), values.end());
        values.resize(unique(values.begin(), values.end()) - values.begin());
        n = (int) values.size();
        init(n, n);
        for(int i = 0; i < n; i++){
            for(int j = 0; j < n; j++){
                if(i == j) continue;
                if((values[j] % values[i]) == 0) addEdge(i, j);
            }
        }
        int max_antichain = n - maxMatching(), ans = max_antichain;
        vector < int > in_sol;
        set < int > result;
        for(int i = 0; i < n; i++) result.insert(values[i]);
        while(!result.empty()){
            set < int > :: iterator it = result.begin();
            int check_val = *it;
            vector < int > tmp; n = 0;
            while((++it) != result.end()){
                if(*it % check_val){
                    ++n;
                    tmp.push_back(*it);
                }
            }
            init(n, n);
            for(int i = 0; i < n; i++){
                for(int j = 0; j < n; j++){
                    if(i == j) continue;
                    if(tmp[j] % tmp[i] == 0) addEdge(i, j);
                }
            }
            if(n - maxMatching() == max_antichain - 1){
                max_antichain = max_antichain - 1;
                in_sol.push_back(check_val);
                set < int > :: iterator it2 = result.begin();
                while((++it2) != result.end()){
                    if(*it2 % check_val == 0){
                        set < int > :: iterator it3 = it2;
                        it3++;
                        result.erase(it2);
                        it2 = (--it3);
                    }
                }
            }
            result.erase(result.begin());
        }
        printf("Case %d:", qq);
        for(int i = 0; i < ans; i++) printf(" %d", in_sol[i]);
        printf("\n");
    }
}
```

2.3.2 Min Path Cover General

```
// LIGHTOJ
#include "bits/stdc++.h"
using namespace std;

const int N = 1005;
const int M = N * N;

/*
   In this problem we need to find a min. path cover in a general dir.graph, with no
   requirement of vertex-disjointedness. Hence, just compress the directed graph
   into a dag by condensing scc's to a single node. After this, perform transitive closure
   on the dag.

Finally : min path cover = count_scc - max_matching()

Note that if vertex disjointedness was required, we would NOT do a transitive closure.
 */

/*
   Hopcroft Karp Max Matching in O(E * sqrt(V))
   N = Number of Nodes, M = Number of Edges
   n1 = Size of left partite, n2 = Size of right partite
   Nodes are numbered from [0, n1 - 1] and [0, n2 - 1]

   init(n1, n2) declares the two partite sizes and resets arrays
   addEdge(x, y) adds an edge between x in left partite and y in right partite
   maxMatching() returns the maximum matching

   Maximum Matching = Minimum Vertex Cover (Konig's Theorem)
   N - Maximum Matching = Maximal Independent Set
 */

int n1, n2, edges, last[N], previous[M], head[M];
int matching[N], dist[N], Q[N];
bool used[N], vis[N];

inline void init(int _n1, int _n2) {
    n1 = _n1;
    n2 = _n2;
    edges = 0;
    fill(last, last + n1, -1);
}

inline void addEdge(int u, int v) {
    head[edges] = v;
    previous[edges] = last[u];
    last[u] = edges++;
}

inline void bfs() {
    fill(dist, dist + n1, -1);
    int sizeQ = 0;
    for(int u = 0; u < n1; ++u){
        if(!used[u]){
            Q[sizeQ++] = u;
            dist[u] = 0;
        }
    }
    for(int i = 0; i < sizeQ; i++){
        int u1 = Q[i];
        for(int e = last[u1]; e >= 0; e = previous[e]){
            int u2 = matching[head[e]];
            if(u2 >= 0 && dist[u2] < 0){
                dist[u2] = dist[u1] + 1;
                Q[sizeQ++] = u2;
            }
        }
    }
}

inline bool dfs(int u1) {
    vis[u1] = true;
    for(int e = last[u1]; e >= 0; e = previous[e]){
        int v = head[e];
        int u2 = matching[v];
        if(u2 < 0 || !vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2)){
            matching[v] = u1;
            used[u1] = true;
            return true;
        }
    }
    return false;
}

inline int maxMatching() {
    fill(used, used + n1, false);
    fill(matching, matching + n2, -1);
    for(int res = 0; ;){
        bfs();
        fill(vis, vis + n1, false);
        int f = 0;
        for(int u = 0; u < n1; ++u)
            if(!used[u] && dfs(u))
                ++f;
        if(!f) return res;
        res += f;
    }
}

// End of Hopcroft Karp Template

int t, n, m, scc;
int visit[N], component[N];
vector < int > adj[N], tra[N], dag[N];
stack < int > order;
```

```
inline void dfs1(int u){
    for(int i = 0; i < (int) adj[u].size(); i++){
        int v = adj[u][i];
        if(!visit[v]){
            visit[v] = true;
            dfs1(v);
        }
    }
    order.push(u);
}

inline void dfs2(int u){
    component[u] = scc;
    for(int i = 0; i < (int) tra[u].size(); i++){
        int v = tra[u][i];
        if(!component[v]) dfs2(v);
    }
}

inline void explore(int root){
    queue < int > bfs;
    bfs.push(root);
    while(!bfs.empty()){
        int node = bfs.front();
        bfs.pop();
        if(node != root) addEdge(root - 1, node - 1);
        for(int i = 0; i < (int) dag[node].size(); i++){
            int next = dag[node][i];
            bfs.push(next);
        }
    }
}

int main(){
    freopen("ioi.in", "r", stdin);
    cin >> t;
    for(int qq = 1; qq <= t; qq++){
        cin >> n >> m;

        // Cleanup the arrays
        for(int i = 1; i <= n; i++){
            adj[i].clear();
            tra[i].clear();
            dag[i].clear();
            visit[i] = false;
            component[i] = 0;
        }
        while(!order.empty()) order.pop();
        scc = 0;

        // Make graph and its transpose
        for(int i = 1; i <= m; i++){
            int u, v;
            cin >> u >> v;
            adj[u].push_back(v);
            tra[v].push_back(u);
        }

        // Kosaraju Phase 1
        for(int i = 1; i <= n; i++){
            if(!visit[i]){
                visit[i] = true;
                dfs1(i);
            }
        }

        // Kosaraju Phase 2
        while(!order.empty()){
            int node = order.top();
            order.pop();
            if(!component[node]){
                ++scc;
                dfs2(node);
            }
        }

        // Add edges between scc'sO
        for(int u = 1; u <= n; u++){
            for(int i = 0; i < (int) adj[u].size(); i++){
                int v = adj[u][i];
                if(component[u] != component[v])
                    dag[component[u]].push_back(component[v]);
            }
        }

        // Remove Multiple Edges : Important step in SCC Condensation!
        for(int i = 1; i <= scc; i++){
            sort(dag[i].begin(), dag[i].end());
            dag[i].resize(unique(dag[i].begin(), dag[i].end()) - dag[i].begin());
        }

        init(scc, scc); // Initialise Bipartite graph with 'count_scc' nodes
        for(int i = 1; i <= scc; i++) explore(i); // Transitive Closure

        int ans = scc - maxMatching(); // Minimum Path Cover
        cout << "Case " << qq << ": " << ans << "\n";
    }
}
```

2.4 General Graph Matching

2.4.1 Edmonds Matching

```
/*

   All credit goes to Alex Li.

   Given any directed graph, determine a maximal subset of its edges such that no
   node is shared between different edges in the resulting subset. edmonds()
   applies to a global, pre-populated adjacency list adj[] which must only consist
   of nodes numbered with integers between 0 (inclusive) and the total number of
   nodes (exclusive), as passed in the function argument.
   Time Complexity:
   - O(n^3) per call to edmonds(), where n is the number of nodes.
   Space Complexity:
   - O(max(n, m)) for storage of the graph, where n the number of nodes and m is
   the number of edges.
   - O(n) auxiliary heap space for edmonds(), where n is the number of nodes.
 */

#include <queue>
#include <vector>
```

```
const int MAXN = 100;
std::vector<int> adj[MAXN];
int p[MAXN], base[MAXN], match[MAXN];

int lca(int nodes, int u, int v) {
    std::vector<bool> used(nodes);
    for (;;) {
        u = base[u];
        used[u] = true;
        if (match[u] == -1) {
            break;
        }
        u = p[match[u]];
    }
    for (;;) {
        v = base[v];
        if (used[v]) {
            return v;
        }
        v = p[match[v]];
    }
}

void mark_path(std::vector<bool> &blossom, int u, int b, int child) {
    for (; base[u] != b; u = p[match[u]]) {
        blossom[base[u]] = true;
        blossom[base[match[u]]] = true;
        p[u] = child;
        child = match[u];
    }
}

int find_path(int nodes, int root) {
    std::vector<bool> used(nodes);
    for (int i = 0; i < nodes; ++i) {
        p[i] = -1;
        base[i] = i;
    }
    used[root] = true;
    std::queue<int> q;
    q.push(root);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int j = 0; j < (int)adj[u].size(); j++) {
            int v = adj[u][j];
            if (base[u] == base[v] || match[u] == v) {
                continue;
            }
            if (v == root || (match[v] != -1 && p[match[v]] != -1)) {
                int curr_base = lca(nodes, u, v);
                std::vector<bool> blossom(nodes);
                mark_path(blossom, u, curr_base, v);
                mark_path(blossom, v, curr_base, u);
                for (int i = 0; i < nodes; i++) {
                    if (blossom[base[i]]) {
                        base[i] = curr_base;
                        if (!used[i]) {
                            used[i] = true;
                            q.push(i);
                        }
                    }
                }
            } else if (p[v] == -1) {
                p[v] = u;
                if (match[v] == -1) {
                    return v;
                }
                v = match[v];
                used[v] = true;
                q.push(v);
            }
        }
    }
    return -1;
}

int edmonds(int nodes) {
    for (int i = 0; i < nodes; i++) {
        match[i] = -1;
    }
    for (int i = 0; i < nodes; i++) {
        if (match[i] == -1) {
            int u, pu, ppu;
            for (u = find_path(nodes, i); u != -1; u = ppu) {
                pu = p[u];
                ppu = match[pu];
                match[u] = pu;
                match[pu] = u;
            }
        }
    }
    int matches = 0;
    for (int i = 0; i < nodes; i++) {
        if (match[i] != -1) {
            matches++;
        }
    }
    return matches/2;
}

/***
  Example Usage and Output:
  Matched 2 pair(s):
  0 1
  2 3
 ***/

#include <iostream>
using namespace std;

int main() {
    int n; cin >> n;
    int u, v;
    while (cin >> u >> v) {
        u--; v--;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    cout << edmonds(n) * 2 << endl;
    for (int i = 0; i < n; i++) {
        if (match[i] != -1 && i < match[i]) {
            cout << i + 1 << " " << match[i] + 1 << endl;
        }
    }
}
```

2.5 Max Flow and Min Cut

2.5.1 Dinics

```
// UKIEPC 2017 Problem K
#include "bits/stdc++.h"
using namespace std;


// Dinic's Algorithm to compute Maximum Flow

const int INF = 1000000000;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic {
    int N;
    vector < vector < Edge > > G;
    vector < Edge * > dad;
    vector < int > Q;

    Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

    void AddEdge(int from, int to, int cap) {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    long long BlockingFlow(int s, int t) {
        fill(dad.begin(), dad.end(), (Edge *) NULL);
        dad[s] = &G[0][0] - 1;

        int head = 0, tail = 0;
        Q[tail++] = s;

        while(head < tail){
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++){
                Edge &e = G[x][i];
                if(!dad[e.to] && e.cap - e.flow > 0){
                    dad[e.to] = &G[x][i];
                    Q[tail++] = e.to;
                }
            }
        }
        if (!dad[t]) return 0;

        long long totflow = 0;
        for (int i = 0; i < G[t].size(); i++){
            Edge *start = &G[G[t][i].to][G[t][i].index];
            int amt = INF;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]){
                if (!e) { amt = 0; break; }
                amt = min(amt, e->cap - e->flow);
            }
            if (amt == 0) continue;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                e->flow += amt;
                G[e->to][e->index].flow -= amt;
            }
            totflow += amt;
        }
        return totflow;
    }

    long long GetMaxFlow(int s, int t) {
        long long totflow = 0;
        while (long long flow = BlockingFlow(s, t)) totflow += flow;
        return totflow;
    }
};

// End of Maximum Flow Template

const int N = 105;
const int M = N * N;

int n, m;
int w[N], l[N], t[N];
int in_crane[N];
int out_crane[N];
int in_building[N];
int out_building[N];
int source, sink, timer;
int par[M];
int rev[M]; // id of crane or building that we're dealing with

inline void solve(int id) {
    int st = out_building[id];
    vector < int > indices;
    while (st != source) {
        indices.push_back(rev[st]);
        st = par[st];
    }
    reverse(indices.begin(), indices.end());
    set < int > printed;
    for (int x: indices) {
        if ((x >= 1 && x <= n) && !printed.count(x)) {
            cout << x << ' ';
            printed.insert(x);
        }
    }
    cout << endl;
}

void bfs(Dinic flow) {
    queue < int > q;
    fill(par, par + M, -1);
    par[source] = source;
    q.push(source);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (Edge e: flow.G[u]) {
            int v = e.to;
            if (e.flow == 1) {
                if (par[v] == -1) {
                    par[v] = u;
                    q.push(v);
                }
            }
        }
    }
    for (int i = 1; i <= m; i++) {
        solve(i);
    }
}

int main() {
    source = 0;
    cin >> n;
    rev[source] = 0;
    for (int i = 1; i <= n; i++) {
        cin >> w[i] >> l[i];
        in_crane[i] = ++timer;
        rev[timer] = i;
        out_crane[i] = ++timer;
        rev[timer] = i;
    }
    cin >> m;
    for (int i = 1; i <= m; i++) {
        cin >> t[i];
        in_building[i] = ++timer;
        rev[timer] = -1;
        out_building[i] = ++timer;
        rev[timer] = -1;
    }
    rev[sink] = 0;
    sink = ++timer;
    Dinic flow(timer + 10);
    for (int i = 1; i <= n; i++) {
        if (w[i] == 0) {
            // source to 0 weight cranes
            flow.AddEdge(source, in_crane[i], 1);
        }
        // each crane has a capacity of 1 -> ensuring vertex disjoint paths
        flow.AddEdge(in_crane[i], out_crane[i], 1);
        for (int j = 1; j <= n; j++) {
            if (j == i) {
                continue;
            }
            if (l[i] >= w[j]) {
                // edge from one crane to another crane.
                flow.AddEdge(out_crane[i], in_crane[j], 1);
            }
        }
        for (int j = 1; j <= m; j++) {
            if (l[i] >= t[j]) {
                // ith crane satisfied jth building
                flow.AddEdge(out_crane[i], in_building[j], 1);
            }
        }
    }
    for (int i = 1; i <= m; i++) {
        // limit outflow from each building to be 1
        flow.AddEdge(in_building[i], out_building[i], 1);
        // add edge to sink
        flow.AddEdge(out_building[i], sink, 1);
    }
    int total = flow.GetMaxFlow(source, sink);
    if (total < m) {
        cout << "impossible" << endl;
    } else {
        assert (total == m);
        bfs(flow);
    }
}
```

2.5.2 Min Cut

```
// SWERC 2015 - Landscaping

/**
  General strategies to solve min-cut problems:

  1) Convert everything to "penalties". Most often the problem
  would reduce to a bipartite graph with a source connecting to
  nodes on the left partite and a sink connected to nodes on the
  right partite.

  2) Don't think in terms of flow, as most often it won't make sense
  straightaway. Think in terms of "cuts" where removing each edge
  can be thought of as a penalty. We want to minimize our penalty
  i.e. find a minimum cut.

  3) If certain constraint must be satisfied, say node (i) on the left
  and node (j) on the right must always be in the "same set", then we
  can model this by adding an infinite edge between these two nodes,
  since infinite edges will never be part of a min-cut. Other constraints
  can be handled using similar tricks.

  Retrieving min-cut from flow network:

  Do a BFS in the final residual graph from the source vertex to compute
  S = set of vertices reachable from source. The edges in the min cut are
  precisely those with one end point in S and the other in V \ S.
 **/

#include "bits/stdc++.h"
using namespace std;

// Dinic's Algorithm to compute Maximum Flow

const int INF = 1000000000;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic {
    int N;
    vector < vector < Edge > > G;
    vector < Edge * > dad;
    vector < int > Q;

    Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

    void AddEdge(int from, int to, int cap) {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    long long BlockingFlow(int s, int t) {
        fill(dad.begin(), dad.end(), (Edge *) NULL);
        dad[s] = &G[0][0] - 1;

        int head = 0, tail = 0;
        Q[tail++] = s;

        while(head < tail){
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++){
                Edge &e = G[x][i];
                if(!dad[e.to] && e.cap - e.flow > 0){
                    dad[e.to] = &G[x][i];
                    Q[tail++] = e.to;
                }
            }
        }
        if (!dad[t]) return 0;

        long long totflow = 0;
        for (int i = 0; i < G[t].size(); i++){
            Edge *start = &G[G[t][i].to][G[t][i].index];
            int amt = INF;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]){
                if (!e) { amt = 0; break; }
                amt = min(amt, e->cap - e->flow);
            }
            if (amt == 0) continue;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from]) {
                e->flow += amt;
                G[e->to][e->index].flow -= amt;
            }
            totflow += amt;
        }
        return totflow;
    }

    long long GetMaxFlow(int s, int t) {
        long long totflow = 0;
        while (long long flow = BlockingFlow(s, t)) totflow += flow;
        return totflow;
    }
};

// End of Maximum Flow Template

const int N = 100;

int n, m, a, b;
int dx[] = {-1, 0, 1, 0};
int dy[] = {0, 1, 0, -1};
int timer;
int id[N][N];
bool high[N][N];

int main() {
    scanf("%d %d %d %d", &n, &m, &a, &b);
    Dinic mf(n * m + 2);
    int source = 0;
    int sink = n * m + 1;
    for (int i = 1; i <= n; i++) {
        char inp[m + 1];
        scanf("%s", inp + 1);
        for (int j = 1; j <= m; j++) {
            id[i][j] = ++timer;
            high[i][j] = (inp[j] == '#');
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (high[i][j]) {
                mf.AddEdge(source, id[i][j], b);
            } else {
                mf.AddEdge(id[i][j], sink, b);
            }
            for (int k = 0; k < 4; k++) {
                int ni = i + dx[k];
                int nj = j + dy[k];
                if (ni >= 1 && ni <= n && nj >= 1 && nj <= m) {
                    mf.AddEdge(id[i][j], id[ni][nj], a);
                }
            }
        }
    }
    cout << mf.GetMaxFlow(source, sink) << endl;
}
```

2.6 Min Cost Max Flow

2.6.1 Mcmf Bellman Ford

```
// LIGHTOJ
#include "bits/stdc++.h"
using namespace std;


// Min cost Max flow template

struct MinimumCostMaximumFlow {

    typedef long long Flow;
    typedef long long Cost;
    static const Cost infiniteDistance = 1e18;
    static const Cost EPS = 1e-7;
    static const Flow infiniteFlow = 1e18;

    struct Edge{
        int u, v;
        Flow f, c;
        Cost w;
        Edge(int u, int v, Flow f, Flow c, Cost w) : u(u), v(v), f(f), c(c), w(w) {}
    };
```

```
vector < Edge > e;
vector < vector < int > > g;
int n, source, sink, *prev;
Cost *dist;

MinimumCostMaximumFlow(int n) : n(n){
    dist = (Cost*)malloc(sizeof(Cost)*n);
    prev = (int*) malloc(sizeof(int)*n);
    g.resize(n);
}

~MinimumCostMaximumFlow(){
    free(dist);
    free(prev);
    g.clear();
}

inline void add(int u, int v, Flow c, Cost w){
    g[u].push_back(e.size());
    e.push_back(Edge(u, v, 0, c, w));
    // For residual graph
    g[v].push_back(e.size());
    e.push_back(Edge(v, u, 0, 0, -w));
}

inline pair < Cost, Flow > getMaxFlow(int source, int sink){
    this -> source = source;
    this -> sink = sink;
    for(int i = 0; i < (int) e.size(); i++) e[i].f = 0;
    Flow flow = 0;
    Cost cost = 0;
    while(bellmanFord()){
        int u = sink;
        Flow pushed = infiniteFlow;
        Cost pushCost = 0;
        while(u != source){
            int id = prev[u];
            pushed = min(pushed, e[id].c - e[id].f);
            pushCost += e[id].w;
            u = e[id].u;
        }
        u = sink;
        while(u != source){
            int id = prev[u];
            e[id].f += pushed;
            e[id ^ 1].f -= pushed;
            u = e[id].u;
        }
        flow += pushed;
        cost += pushCost * pushed;
    }
    return make_pair(cost, flow);
}

inline bool bellmanFord(){
    for(int i = 0; i < n; ++i) dist[i] = infiniteDistance;
    dist[source] = 0;
    for(int k = 0; k < n; ++k){
        bool update = false;
        for(int id = 0; id < (int) e.size(); ++id){
            int u = e[id].u;
```

```
                int v = e[id].v;
                if(dist[u] + EPS >= infiniteDistance) continue;
                Cost w = e[id].w;
                if(e[id].f < e[id].c && dist[v] > dist[u] + w + EPS){
                    dist[v] = dist[u] + w;
                    prev[v] = id;
                    update = true;
                }
            }
            if(!update) break;
        }
        return (dist[sink] + EPS) < (infiniteDistance);
    }

    // After running mcmf, e[id].f has the flow which has passed through that edge in the optimal soln
    inline void displayEdges(){
        cout << "******" << '\n';
        for(int i = 0; i < (int) e.size(); ++i)
            cout << e[i].u << " " << e[i].v << " " << e[i].f << " " << e[i].c << " " << e[i].w <<"\n";
        cout << "******" << '\n';
    }
};




const int N = 1e2 + 2;

int t, n, m;
int a[N][N], in[N][N], out[N][N];

inline bool is_valid(int x, int y){
    return (x >= 1 and x <= n and y >= 1 and y <= m);
}

int main(){
    freopen("ioi.in", "r", stdin);
    cin >> t;
    for(int qq = 1; qq <= t; qq++){
        cin >> n >> m;
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= m; j++)
                cin >> a[i][j];
        int cur_time = 0;
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                in[i][j]  = ++cur_time;
                out[i][j] = in[i][j] + (n * m);
            }
        }
        MinimumCostMaximumFlow mcmf(2 * n * m + 1);
        int source = out[1][1], sink = in[n][m];
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){
                mcmf.add(in[i][j], out[i][j], 1, -a[i][j]);
                if(is_valid(i, j + 1))
                    mcmf.add(out[i][j], in[i][j + 1], 1, 0);
                if(is_valid(i + 1, j))
                    mcmf.add(out[i][j], in[i + 1][j], 1, 0);
            }
        }
        cout << "Case " << qq  << ": " << (a[1][1] + a[n][m] -mcmf.getMaxFlow(source, sink).first);
        cout << "\n";
    }
}
```

2.6.2 Mcmf SPFA

```
// Find T Paths from (1, 1) to (n, m) such that the sum of the SET of values in the union of the T paths is
    maximized.
// We can also handle Disjointedness of Paths easily.
// This assumes A_{ij} is positive.

#include "bits/stdc++.h"
using namespace std;

// Min cost Max flow Template using SPFA

struct Edge {
    int u, v;
    long long cap, cost;

    Edge(int _u, int _v, long long _cap, long long _cost) {
        u = _u; v = _v; cap = _cap; cost = _cost;
    }
};

struct MinimumCostMaximumFlow{
    int n, s, t;
    long long flow, cost;
    vector<vector<int> > graph;
    vector<Edge> e;
    vector<long long> dist;
    vector<int> parent;

    MinimumCostMaximumFlow(int _n){
        // 0-based indexing
        n = _n;
        graph.assign(n, vector<int> ());
    }

    void add(int u, int v, long long cap, long long cost, bool directed = true){
        graph[u].push_back(e.size());
        e.push_back(Edge(u, v, cap, cost));

        graph[v].push_back(e.size());
        e.push_back(Edge(v, u, 0, -cost));

        if(!directed)
            add(v, u, cap, cost, true);
    }

    pair<long long, long long> getMinCostFlow(int _s, int _t){
        s = _s; t = _t;
        flow = 0, cost = 0;

        while(SPFA()){
            flow += sendFlow(t, 1LL<<62);
        }

        return make_pair(flow, cost);
    }
    // Not sure about negative cycle
    bool SPFA(){
        parent.assign(n, -1);
        dist.assign(n, 1LL<<62);        dist[s] = 0;
        vector<int> queuetime(n, 0);    queuetime[s] = 1;
        vector<bool> inqueue(n, 0);     inqueue[s] = true;
        queue<int> q;                   q.push(s);
        bool negativecycle = false;


        while(!q.empty() && !negativecycle){
            int u = q.front(); q.pop(); inqueue[u] = false;

            for(int i = 0; i < graph[u].size(); i++){
                int eIdx = graph[u][i];
                int v = e[eIdx].v, w = e[eIdx].cost, cap = e[eIdx].cap;

                if(dist[u] + w < dist[v] && cap > 0){
                    dist[v] = dist[u] + w;
                    parent[v] = eIdx;

                    if(!inqueue[v]){
                        q.push(v);
                        queuetime[v]++;
                        inqueue[v] = true;

                        if(queuetime[v] == n+2){
                            negativecycle = true;
                            break;
                        }
                    }
                }
            }
        }

        return dist[t] != (1LL<<62);
    }

    long long sendFlow(int v, long long curFlow){
        if(parent[v] == -1)
            return curFlow;
        int eIdx = parent[v];
        int u = e[eIdx].u, w = e[eIdx].cost;

        long long f = sendFlow(u, min(curFlow, e[eIdx].cap));

        cost += f*w;
        e[eIdx].cap -= f;
        e[eIdx^1].cap += f;

        return f;
    }
};

/***** End of Minimum Cost Maximum Flow Template *****/

const int N = 105;

int n, m, t, id;
int arr[N][N];
int in[N][N], out[N][N];
int main() {
    cin >> n >> m >> t;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> arr[i][j];
        }
    }
    // Enumerate the nodes.
    int source = ++id;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            in[i][j] = ++id;
            out[i][j] = ++id;
        }
    }
    int sink = ++id;
    // Add appropriate edges
    MinimumCostMaximumFlow mcmf(id + 10);
    mcmf.add(source, in[1][1], t, 0);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            mcmf.add(in[i][j], out[i][j], 1, -arr[i][j]); // The first time we take this edge we add its value
            mcmf.add(in[i][j], out[i][j], t - 1, 0); // Subsequent times we don't add its value
            // If we want to make paths disjoint we can just not have the above edge at all.
            if (i + 1 <= n) {
                mcmf.add(out[i][j], in[i + 1][j], t, 0);
            }
            if (j + 1 <= m) {
                mcmf.add(out[i][j], in[i][j + 1], t, 0);
            }
        }
    }
    mcmf.add(out[n][m], sink, t, 0);
    cout << -mcmf.getMinCostFlow(source, sink).second << endl;
}
```

2.6.3 Mcmf Trees

```
// LightOJ
#include "bits/stdc++.h"
using namespace std;


// Min cost Max flow template

struct MinimumCostMaximumFlow {

    typedef long long Flow;
    typedef long long Cost;
    static const Cost infiniteDistance = 1e18;
    static const Cost EPS = 1e-7;
    static const Flow infiniteFlow = 1e18;

    struct Edge{
        int u, v;
        Flow mx, c;
        Cost w;
        Edge(int u, int v, Flow mx, Flow c, Cost w) : u(u), v(v), mx(mx), c(c), w(w) {}
    };
```

```
vector < Edge > e;
vector < vector < int > > g;
int n, source, sink, *prev;
Cost *dist;

MinimumCostMaximumFlow(int n) : n(n){
    dist = (Cost*)malloc(sizeof(Cost)*n);
    prev = (int*) malloc(sizeof(int)*n);
    g.resize(n);
}

~MinimumCostMaximumFlow(){
    free(dist);
    free(prev);
    g.clear();
}

inline void add(int u, int v, Flow c, Cost w){
    g[u].push_back(e.size())    ;
    e.push_back(Edge(u, v, 0, c, w));
    // For residual graph
    g[v].push_back(e.size());
    e.push_back(Edge(v, u, 0, 0, -w));
}

inline pair < Cost, Flow > getMaxFlow(int source, int sink){
    this -> source = source;
    this -> sink = sink;
    for(int i = 0; i < (int) e.size(); i++) e[i].mx = 0;
    Flow flow = 0;
    Cost cost = 0;
    while(bellmanFord()){
        int u = sink;
        Flow pushed = infiniteFlow;
        Cost pushCost = 0;
        while(u != source){
            int id = prev[u];
            pushed = min(pushed, e[id].c - e[id].mx);
            pushCost += e[id].w;
            u = e[id].u;
        }
        u = sink;
        while(u != source){
            int id = prev[u];
            e[id].mx += pushed;
            e[id ^ 1].mx -= pushed;
            u = e[id].u;
        }
        flow += pushed;
        cost += pushCost * pushed;
    }
    return make_pair(cost, flow);
}

inline bool bellmanFord(){
    for(int i = 0; i < n; ++i) dist[i] = infiniteDistance;
    dist[source] = 0;
    for(int k = 0; k < n; ++k){
        bool update = false;
        for(int id = 0; id < (int) e.size(); ++id){
            int u = e[id].u;
```

```
                int v = e[id].v;
                if(dist[u] + EPS >= infiniteDistance) continue;
                Cost w = e[id].w;
                if(e[id].mx < e[id].c && dist[v] > dist[u] + w + EPS){
                    dist[v] = dist[u] + w;
                    prev[v] = id;
                    update = true;
                }
            }
            if(!update) break;
        }
        return (dist[sink] + EPS) < (infiniteDistance);
    }

    // After running mcmf_max, e[id].mx has the flow which has passed through that edge in the optimal soln
    inline void displayEdges(){
        cout << "******" << '\n';
        for(int i = 0; i < (int) e.size(); ++i)
            cout << e[i].u << " " << e[i].v << " " << e[i].mx << " " << e[i].c << " " << e[i].w <<"\n";
        cout << "******" << '\n';
    }
};

/*
   In this problem, we are given 2 rooted trees and asked : What is the minimum number of leaves
   you can add to either of these trees such that they are isomorphic i.e. structurally similar?

   We will do the following dp :
   dp[u][v] = Minimum number of nodes to add such that subtree rooted at (u) in T1 and subtree
   rooted at (v) in T2 are isomorphic.

Answer : dp[0][0]

Base Cases : If (u) and (v) are both leaves, dp[u][v] = 0

Recurrence :-
Let u have n children c1, c2...cn, and v have m children d1, d2, d3.... dm.

You can match any c_i with any d_j with cost dp[c_i][d_j].
You need to match all c_i with some d_j.

Notice that if m > n, some c_is won't be matched, so you must add that entire subtree
as it is. Similarly, if n > m, some d_j's won't be matched hence you must add the entire
subtree as it is to make it isomorphic. This can be handled by taking max(n, m) nodes in
each partite (thus one of the partites will have some null nodes). The cost of edges with
one end point as a null node will be the subtree size of the node at to the other endpoint.

After making this bipartite graph, you can run min-cost-max-flow to find the minimum
cost matching and that would be the desired dp[u][v] value.

Complexity : O(N ** 3 log N). Summation of edges added = O(N ** 2), and since mcmf works in
O(V * E * log E), complexity equals O(N ** 3 log N)
 */

const int N = 200;
int t, n1, n2;
int sub1[N], sub2[N], dp[N][N];
vector < int > adj1[N], adj2[N];

inline void dfs1(int u){
    sub1[u] = 1;
    for(int i = 0; i < (int) adj1[u].size(); i++){
        int v = adj1[u][i];
        dfs1(v);
        sub1[u] += sub1[v];
    }
}

inline void dfs2(int u){
    sub2[u] = 1;
    for(int i = 0; i < (int) adj2[u].size(); i++){
        int v = adj2[u][i];
        dfs2(v);
        sub2[u] += sub2[v];
    }
}

inline int solve(int u, int v){
    if(dp[u][v] != -1) return dp[u][v];
    int ucnt = adj1[u].size(), vcnt = adj2[v].size();
    int match = max(ucnt, vcnt); // Number of nodes in each partite
    if(match == 0) return dp[u][v] = 0; // Base Case
    for(int i = 1; i <= match; i++)
        for(int j = 1; j <= match; j++)
            if(i <= ucnt and j <= vcnt) solve(adj1[u][i - 1], adj2[v][j - 1]);
    MinimumCostMaximumFlow mcmf(match * 2 + 2);
    int source = 0, sink = match * 2 + 1;
    for(int i = 1; i <= match; i++) mcmf.add(source, i, 1, 0);
    for(int i = 1; i <= match; i++){
        for(int j = 1; j <= match; j++){
            if(i <= ucnt and j <= vcnt) // Edge between two children
                mcmf.add(i, match + j, 1, dp[adj1[u][i - 1]][adj2[v][j - 1]]);
            else if(i > ucnt) // Edge between null node and proper node
                mcmf.add(i, match + j, 1, sub2[adj2[v][j - 1]]);
            else // Edge between null node and proper node
                mcmf.add(i, match + j, 1, sub1[adj1[u][i - 1]]);
        }
    }
    for(int i = match + 1; i <= match * 2; i++) mcmf.add(i, sink, 1, 0);
    pair < long long, long long > res = mcmf.getMaxFlow(source, sink);
    assert(res.second == match); // All nodes must be paired
    return dp[u][v] = res.first;
}

int main(){
    cin >> t;
    for(int qq = 1; qq <= t; qq++){
        cin >> n1;
        for(int i = 0; i < N; i++){
            adj1[i].clear();
            adj2[i].clear();
            sub1[i] = 0;
            sub2[i] = 0;
        }
        for(int i = 1; i <= n1; i++){
            int p; cin >> p;
            adj1[p].push_back(i);
        }
        cin >> n2;
        for(int i = 1; i <= n2; i++){
            int p; cin >> p;
            adj2[p].push_back(i);
        }
        dfs1(0); dfs2(0);
        memset(dp, -1, sizeof dp);
        int ans = solve(0, 0);
        cout << "Case " << qq << ": " << ans << '\n';
    }
}
```

3 Game Theory

3.1 A Partitioning Game

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

const int N = 100;
const int X = 10000;

/*
   Let's find grundy(x), which is the grundy number for this game on a pile
   with x stones. Note that we can xor grundy(x_i) for each x_i in the test
   case and that would be sufficient to know the answer.

   Now, let's focus on computing grundy(x) for some pile size (x).
   We know that if x <= 2, grundy(x) = 0, as (x) is a losing position.

   Now, let's look at some x > 2, we can split x into (i, j) such that
   (i != j) and (i + j = x).

   Every such pair {i, j} can be treated as two disjoint subgames, one with
   pile size (i) and the other with pile size (j) : Thus the grundy of this
   state would be xor of these two subgames.

   So, For all valid pairs {i, j} add grundy(i) ^ grundy(j) to a set.
   Thus you will have the set of all states reachable from the current state (x).
   Find the mex in that to find grundy(x)
 */

int t, n, arr[N + 5], dp[X + 5];

inline int solve(int x){
    if(x <= 2) return 0;
    if(dp[x] != -1) return dp[x];
    vector < int > values;
    for(int i = 1; i <= x >> 1; i++){
        if(i != (x - i))
            values.push_back(solve(i) ^ solve(x - i));
    }
    sort(values.begin(), values.end());
    values.resize(unique(values.begin(), values.end()) - values.begin());
    int mex = (int) values.size();
    for(int i = 0; i < (int) values.size(); i++){
        if(values[i] != i){
            mex = i;
            break;
        }
    }
    return dp[x] = mex;
}

int main(){
    scanf("%d", &t);
    memset(dp, -1, sizeof dp);
    for(int i = 1; i <= X; i++) dp[i] = solve(i);
    for(int qq = 1; qq <= t; qq++){
        scanf("%d", &n);
        int ans = 0;
        for(int i = 1; i <= n; i++){
            scanf("%d", arr + i);
            ans ^= dp[arr[i]];
        }
        printf("Case %d: ", qq);
        if(ans) puts("Alice");
        else puts("Bob");
    }
}
```

3.2 A String Game

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

/*
   In this problem, a subgame can be considered as a substring of the original string.
   We start with str[1...N], and we can remove any important substring from this and give
   two disjoint substrings to the opponent. The opponent then has the choice to choose
   any one of these two disjoint subgames, hence the value of such a state is simply
   the xor of these two subgames.

   So, the problem reduces to computing the reachable grundy values from the initial
   state and then finding the mex. The mex will give us the grundy value of the original
   string, which would help us determine the answer.
   If the mex is nonzero, we can win. Else we will surely lose.

   Suppose that this game was being played on many strings simultaneously. In that case,
   we would just take the xor of each of the mex values we found and check if it's nonzero.

   It is important to identify the sub-games to solve problems on Grundy Numbers!
 */


const int N = 55;
int t, n, len;
int dp[N][N], exist[N][N];
char str[N], arr[N][N];

inline int solve(int l, int r){
    if(l > r) return 0;
    if(dp[l][r] != -1) return dp[l][r];
    set < int > values;
    for(int i = l; i <= r; i++)
        for(int j = i; j <= r; j++)
            if(exist[i][j])
                values.insert(solve(l, i - 1) ^ solve(j + 1, r));
    int mex = 0;
    while(values.count(mex)) mex++;
    return dp[l][r] = mex;
}

int main(){
    freopen("ioi.in", "r", stdin);
    scanf("%d", &t);
    while(t--){
        scanf("%s", str + 1);
        len = strlen(str + 1);
        scanf("%d", &n);
        for(int i = 1; i <= n; i++){
            scanf("%s", arr[i] + 1);
        }
        memset(exist, 0, sizeof exist);
        memset(dp, -1, sizeof dp);
        for(int i = 1; i <= len; i++){
            for(int j = i; j <= len; j++){
                for(int k = 1; k <= n; k++){
                    if(strlen(arr[k] + 1) != j - i + 1) continue;
                    int ptr = 1;
                    while(ptr <= (j - i + 1) && arr[k][ptr] == str[i + ptr - 1])
                        ptr++;
                    if(ptr > (j - i + 1)) exist[i][j] = 1;
                }
            }
        }
        if(solve(1, len)) puts("Teddy");
        else puts("Tracy");
    }
}
```

3.3 Prime Game

```
// Map DP to compute Grundy Numbers

#include "bits/stdc++.h"
using namespace std;

map < int, int > prime_mask; // Exponents present for each prime
map < int, int > memo;
set < int > primes;

inline void factorize(int x) {
    for (int p = 2; p * p <= x; p++) {
        int cnt = 0;
        while ((x % p) == 0) {
            x /= p;
            cnt += 1;
        }
        if (cnt > 0) {
            prime_mask[p] |= (1 << cnt);
            primes.insert(p);
        }
    }
    if (x != 1) {
        prime_mask[x] |= (1 << 1);
        primes.insert(x);
    }
}

inline int grundy(int mask) {
    if (mask == 0) {
        return memo[mask] = 0;
    }
    if (memo.count(mask)) {
        return memo[mask];
    }
    set < int > reachable_states;
    for (int k = 1; k <= 30; k++) {
        int new_mask = 0;
        for (int i = 1; i <= 30; i++) {
            if ((mask >> i) & 1) {
                if (i < k) {
                    new_mask |= (1 << i);
                } else if (i > k) {
                    new_mask |= (1 << (i - k));
                }
            }
        }
        if (new_mask != mask) {
            reachable_states.insert(grundy(new_mask));
        }
    }
    int mex = 0;
    while (reachable_states.count(mex)) {
        mex += 1;
    }
    return memo[mask] = mex;
}

int main() {
    ios :: sync_with_stdio(false);
    int n; cin >> n;
    for (int i = 1; i <= n; i++) {
        int x; cin >> x;
        factorize(x);
    }
    int res = 0;
    for (int prime_factor: primes) {
        memo.clear();
        res ^= grundy(prime_mask[prime_factor]);
    }
    if (not res) {
        cout << "Arpa" << endl;
    } else {
        cout << "Mojtaba" << endl;
    }
}
```

4 Geometry

4.1 3D

4.1.1 3D Template

```
#include <iostream>
#include <ctime>
#include <fstream>
#include <cmath>
#include <cstring>
#include <cassert>
#include <cstdio>
#include <algorithm>
#include <iomanip>
```

```
#include <vector>
#include <stack>
#include <queue>
#include <set>
#include <map>
#include <complex>
#include <utility>
#include <cctype>
#include <list>
#include <bitset>
#include <unordered_set>
#include <unordered_map>

using namespace std;

#define FORALL(i,a,b) for(int i=(a);i<=(b);++i)
#define FOR(i,n) for(int i=0;i<(n);++i)
#define FORB(i,a,b) for(int i=(a);i>=(b);--i)

typedef long long ll;
typedef long double ld;

typedef pair<ll,int> plli;
typedef pair<int,int> pii;
typedef map<int,int> mii;

#define pb push_back
#define mp make_pair
#define A first
#define B second

#define EPS (1e-8)
#define MAXN 1005
#define sign(x) (((x)>EPS)-((x)<(-EPS)))

const ld PI = atan2(0,-1);

// 3d vector (can degenerate to 2d when z=0)
#define T double
struct vec {
    T x,y,z;    //coordinates/data
    vec(T xx, T yy, T zz=0.){ x=xx;y=yy;z=zz; }
    vec() { x=y=z=0;}

    // vector ops
    vec& operator=(const vec& b) { x=b.x; y=b.y; z=b.z; return *this; }
    vec operator+(const vec& b) const { return vec(x+b.x, y+b.y, z+b.z); }
    vec operator-(const vec& b) const { return vec(x-b.x, y-b.y, z-b.z); }
    T operator*(const vec& b) const { return x*b.x + y*b.y + z*b.z; }
    vec operator^(const vec& b) const { return vec(y*b.z - z*b.y,
            z*b.x - x*b.z,
            x*b.y - y*b.x); }
    // scalar mult
    vec operator*(T k) const { return vec(x*k,y*k,z*k); }
    vec operator/(T k) const { return vec(x/k,y/k,z/k); }
    vec operator-() const { return vec(-x,-y,-z); }  // negation

    T sqlen() const { return (*this) * (*this); }

    bool operator<(const vec& other) const {
        if (x < other.x) return true;
        if (x > other.x) return false;
        if (y < other.y) return true;
        if (y > other.y) return false;
        if (z < other.z) return true;
        if (z > other.z) return false;
        return false;
    }
};
vec operator*(T k, vec v) { return v*k; }
ostream& operator<<(ostream& out, const vec& v) {
    return out << "(" << v.x << "," << v.y << "," << v.z <<")";
}

double dot(const vec& a, const vec& b){
    return a.x * b.x + a.y * b.y + a.z * b.z;
}

#undef T

#define INSIDE (-1)
#define ON (0)
#define OUTSIDE (1)

typedef vector<vec> edge;
typedef vector<vec> face;
typedef vector<face> hull;

ld len(vec a) {
    return sqrtl(a.sqlen());
}

bool eq(ld a, ld b) {
    return abs(b-a) <= EPS;
}

int side(vec a, vec b, vec c, vec x) {
    vec norm = (b-a) ^ (c-a);
    vec me = x-a;
    return sign(me * norm);
}

bool is_colinear(vec a, vec b, vec c) {
    vec u = b-a, v = c-a;
    vec w = u^v;
    return eq(w.sqlen(),0);
}

vec projection(vec a, vec b, vec c, vec x) {
    if (side(a,b,c,x) == ON) return x;
    vec norm = (b-a) ^ (c-a);
    vec ans = x - norm * ((norm * (x-a)) / (norm * norm));
    assert(side(a,b,c,ans) == ON);
    return ans;
}

struct Plane{
    vec r, n;
    Plane(vec r1, vec n1) {
        r = r1;
        n = n1 / len(n1);
    }
    Plane(vec a, vec b, vec c){
        r = a;
        vec qq = (b - a) ^ (c - a);
        n = qq / len(qq);
    }

    //  extraction of form ax + by + cz + d = 0:
    inline double a(){  return n.x; }
    inline double b(){  return n.y; }
    inline double c(){  return n.z; }
    inline double d(){  return -(n.x * r.x + n.y * r.y + n.z * r.z);    }
};

double dihedral_angle(const Plane& p1, const Plane& p2){
    return acos(p1.n * p2.n);
}

hull find_hull(vec* P, int N) {
    random_shuffle(P, P+N);

    // Find 4 non-degenerate points (make a tetrahedron)
    FORALL(j,2,N-1) if (!is_colinear(P[0],P[1],P[j])) { swap(P[j], P[2]); break; }
    FORALL(j,3,N-1) if (side(P[0],P[1],P[2],P[j]) != 0) { swap(P[j], P[3]); break; }

    // Canonicalize them
    if (side(P[0],P[1],P[2],P[3]) == OUTSIDE) swap(P[0], P[1]);
    assert(side(P[0],P[1],P[2],P[3]) == INSIDE);
    assert(side(P[0],P[3],P[1],P[2]) == INSIDE);
    assert(side(P[0],P[2],P[3],P[1]) == INSIDE);
    assert(side(P[3],P[2],P[1],P[0]) == INSIDE);

    hull H{
        {P[0],P[1],P[2]},
            {P[0],P[3],P[1]},
            {P[0],P[2],P[3]},
            {P[3],P[2],P[1]}
    };

    auto make_degrees = [&](const hull& H) {
        map<edge,int> ans;
        for (const auto & f : H) {
            assert(f.size() == 3);
            FOR(i,3) {
                vec a = f[i];
                vec b = f[(i+1)%3];
                ans[{a,b}]++;
            }
        }

        return ans;
    };

    // incrementally add points
    FORALL(j,4,N-1) {
        hull H2; H2.reserve(H.size());
        vector<face> plane;

        for (const auto & f : H) {
            int s = side(f[0],f[1],f[2],P[j]);
            if (s == INSIDE || s == ON) H2.pb(f);
        }

        // For any edge that now only has 1 incident face (it's other face deleted)
        // add a new face with this vertex and that edge.
        map<edge, int> D = make_degrees(H2);
        const auto tmp = H2;
        for (const auto & f : tmp) {
            assert(f.size() == 3);
            FOR(i,3) {
                vec a = f[i];
                vec b = f[(i+1)%3];
                int d = D[{a,b}] + D[{b,a}];
                assert(d == 1 || d == 2);
                if (d==1) {
                    // add a new face
                    H2.pb({a, P[j], b});
                }
            }
        }

        H = H2;
    }

    // sanity check that this is at least mostly a hull :)
    for (const auto & f : H) {
        FOR(i,N) {
            int s = side(f[0],f[1],f[2],P[i]);
            assert(s == INSIDE || s == ON);
        }
    }

    // sanity check that this figure is closed
    map<edge, int> D = make_degrees(H);
    for (const auto & f : H) {
        assert(f.size() == 3);
        FOR(i,3) {
            vec a = f[i];
            vec b = f[(i+1)%3];
            int d = D[{a,b}] + D[{b,a}];
            assert(d == 2);
        }
    }

    return H;
}

//  0: parallel 1: unique line  2: overlapping
typedef pair<vec, vec> Segment;
int intersection_line(const Plane& p1, const Plane& p2, Segment &ans){
    vec u = p1.n ^ p2.n;
    double ax = abs(u.x);
    double ay = abs(u.y);
    double az = abs(u.z);

    if(ax + ay + az < EPS){
        vec v = p2.r - p1.r;
        if(abs(dot(p1.n, v)) < EPS)
            return 2;
        else
            return 0;
    }

    vec iP;
    double d1 = -dot(p1.n, p1.r);
    double d2 = -dot(p2.n, p2.r);
    if(ax >= ay && ax >= az){
        iP.y = (d2 * p1.n.z - d1 * p2.n.z) /  u.x;
        iP.z = (d1 * p2.n.y - d2 * p1.n.y) /  u.x;
    }
    else if(ay >= ax && ay >= az){
        iP.x = (d1 * p2.n.z - d2 * p1.n.z) /  u.y;
        iP.z = (d2 * p1.n.x - d1 * p2.n.x) /  u.y;
    }
    else{
        iP.x = (d2 * p1.n.y - d1 * p2.n.y) /  u.z;
        iP.y = (d1 * p2.n.x - d2 * p1.n.x) /  u.z;
    }

    ans.A = iP;
    ans.B = iP + u;
    return 1;
}

// line stuff
bool on (vec a, vec b, vec x) {
    return eq(len(x-a) + len(x-b), len(a-b));
}

// find the intersection point of ab with cd
vec isect(vec a, vec b, vec c, vec d) {
    vec u = (b-a), v = (d-c), z = (c-a);
    vec vz = v^z, vu = v^u;
    ld s = len(vz) / len(vu) * sign(vz*vu);
    return a + u*s;
}

typedef pair<vec, ld> circle_t;

bool in_circle(const vec& v, const circle_t& C) {
    return len(v - C.first) <= C.second + EPS;
}

circle_t better(circle_t A, circle_t B) {
    if (A.second < B.second) return A;
    return B;
}

circle_t find_circle(vec a) { return {a, 0}; }
circle_t find_circle(vec a, vec b) { return { (a+b)/2, len(a-b) / 2 }; }
circle_t find_circle(vec a, vec b, vec c, bool force_on = false) {
    vec u = (b-a), v = (c-a);
    vec norm = u ^ v;
    vec uperp = u^norm, vperp = v^norm;
    vec ab = (a+b)/2, ac = (a+c)/2;

    if (is_colinear(a,b,c)) {
        if (on(a,b,c)) return { (a+b)/2, len(a-b) / 2 };
        if (on(a,c,b)) return { (a+c)/2, len(a-c) / 2 };
        if (on(c,b,a)) return { (c+b)/2, len(c-b) / 2 };
        assert(false);
    }

    vec ans = isect(ab, ab + uperp, ac, ac + vperp);
    assert(eq(len(ans-a), len(ans-b)));
    assert(eq(len(ans-a), len(ans-c)));

    circle_t C = { ans, (len(ans-a) + len(ans-b) + len(ans-c)) / 3.0l };
    assert(in_circle(a, C) && eq(len(ans-a), C.second));
    assert(in_circle(b, C) && eq(len(ans-b), C.second));
    assert(in_circle(c, C) && eq(len(ans-c), C.second));

    if (force_on) return C;

    circle_t C_ab = find_circle(a,b);
    circle_t C_bc = find_circle(b,c);
    circle_t C_ac = find_circle(a,c);

    if (in_circle(c, C_ab)) C = better(C, C_ab);
    if (in_circle(a, C_bc)) C = better(C, C_bc);
    if (in_circle(b, C_ac)) C = better(C, C_ac);

    assert(in_circle(a,C));
    assert(in_circle(b,C));
    assert(in_circle(c,C));

    return C;
}

// Find circle of N points. K of them (the last K) are guaranteed
// to be on the boundary.
circle_t find_circle(vec* P, int N, int K) {
    if (K >= 3) {
        assert(K == 3);
        assert(!is_colinear(P[N-1],P[N-2],P[N-3]));
        assert(side(P[N-1],P[N-2],P[N-3],P[0]) == ON);
        auto C = find_circle(P[N-1],P[N-2],P[N-3],true);
        return C;
    }

    if (N == 1) return find_circle(P[0]);
    if (N == 2) return find_circle(P[0],P[1]);
    assert(K < N);

    // pick a random point, remove it, recurse.
    // with very high probability, that recursed circle is the optimal circle
    // if not, we just try again (and this point is added to the K set)
    int i = rand()%(N-K);

    swap(P[i], P[N-1-K]); swap(P[N-1-K], P[N-1]); // hack: avoid deleting back K
    auto C = find_circle(P, N-1, K);
    swap(P[N-1-K], P[N-1]); swap(P[i], P[N-1-K]);

    if (in_circle(P[i],C)) return C;

    // Didn't work, that's fine. Add it to our K-set ("boundary set") and try again
    swap(P[i], P[N-1-K]);
    C = find_circle(P, N, K+1);
    swap(P[i], P[N-1-K]);

    return C;
}

int main() {

    Plane p1 (vec(0,0,0), vec(10,0,0), vec(0,10,0));
    Plane p2 (vec(0,0,0), vec(10,0,0), vec(0,0,10));
    Segment intersect;
    if (intersection_line(p1, p2, intersect) == 1) {
        cout << intersect.A << " " << intersect.B << "\n";
    }
}
```

4.2 Circles

4.2.1 Circles

```
typedef pair<PT, double> circle;

// determine if point p is inside circle c.
bool contains(const circle& c, const PT& p) {
    return LE(abs(c.A - p), c.B);
}

//  intersection(s) of line a-b with circle @ c with radius r > 0
vector <PT> line_circle(PT a, PT b, PT c, double r) {
    vector<PT> ret;
    b = b - a;
    a = a - c;
    double A = norm(b);
    double B = dot(a, b);
    double C = norm(a) - r*r;
    double D = B*B - A*C;
    if (D < -eps) return ret;
    ret.push_back(c + a + b*(-B + sqrt(D + eps)) / A);
    if (D > eps)
        ret.push_back(c + a + b*(-B - sqrt(D)) / A);
    return ret;
}

//circle inscribed within points a, b, and c
circle incircle(const PT & a, const PT & b, const PT & c) {
    double al = abs(b - c);
    double bl = abs(a - c);
    double cl = abs(a - b);
    double p = al + bl + cl;
    if (EQ(p, 0)) return circle(a, 0);
    circle res;
    double x = (al * a.X + bl * b.X + cl * c.X) / p;
    double y = (al * a.Y + bl * b.Y + cl * c.Y) / p;
    double r = fabs((a.X - c.X) * (b.Y - c.Y) - (a.Y - c.Y) * (b.X - c.X)) / p;
    return circle(PT(x, y), r);
}

//  compute center of circle given three points
pair<PT, double> circle_center(const PT& a, const PT& b, const PT& c) {
    PT ab = (a + b) / double(2);
    PT ac = (a + c) / double(2);
    PT center;
    line_line(ab, ab + rotate_ccw(a - b, PT(0, 0), M_PI/2), ac, ac + rotate_ccw( a - c, PT(0 ,0), M_PI/2), center);
    return MP(center, abs(center - a));
}

//  compute center of circle given oppsite diametral points a and b.
pair<PT, double> circle_center(const PT& a, const PT& b) {
    PT cntr = (a + b) * (1/2.0);
    double r = abs(a - b) * (1/2.0);
    return MP(cntr, r);
}
```

```
// given two points on the circle and the radius, find the center.
// set flip to true, to get the second circle.
bool circle_center(PT p1, PT p2, double r, PT &c, bool flip) {
    if (flip) swap(p1, p2);
    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);
    double x = (p1.X + p2.X) * 0.5 + (p1.Y - p2.Y) * h;
    double y = (p1.Y + p2.Y) * 0.5 + (p2.X - p1.X) * h;
    c = PT(x, y);
    return true;
}

// find lines that are tangent to both circles.
// lines are external
typedef pair<PT, PT> Segment;
pair<Segment,Segment> circle_tangent(circle a, circle b){
    double R2 = a.B;
    double R1 = b.B;
    PT d = b.A - a.A;
    double theta = acos((R2 - R1) / abs(d));

    Segment a1, a2;
    a1.A = a.A + d/PT(abs(d),0.0)*polar(1.0,theta)*R2;
    a2.A = a.A + d/PT(abs(d),0.0)*polar(1.0,-theta)*R2;

    a1.B = b.A + d/PT(abs(d),0.0)*polar(1.0,theta)*R1;
    a2.B = b.A + d/PT(abs(d),0.0)*polar(1.0,-theta)*R1;
    return make_pair(a1,a2);
}

// find tangent lines to a and b. on the inner side
bool inner_tangent(circle a, circle b, pair<Segment, Segment>& tangents) {
    double d = abs(a.A - b.A);
    double R1 = a.B, R2 = b.B;
    if (d - R1 - R2 < eps) return false;

    double x = d * R1 / (R1 + R2);
    double theta = acos(R1 / x);

    PT p1 = rotate_ccw(b.A, a.A, theta); p1 = a.A + (p1 - a.A) * (R1 / abs(p1 - a.A));
    PT p2 = rotate_ccw(a.A, b.A, theta); p2 = b.A + (p2 - b.A) * (R2 / abs(p2 - b.A));
    tangents.A = MP(p1, p2);

    PT q1 = rotate_ccw(b.A, a.A, -theta); q1 = a.A + (q1 - a.A) * (R1 / abs(q1 - a.A));
    PT q2 = rotate_ccw(a.A, b.A, -theta); q2 = b.A + (q2 - b.A) * (R2 / abs(q2 - b.A));
    tangents.B = MP(q1, q2);
    return true;
}

// find center of a circle of radius R that is tangent to both circles centered
// at a and b with radii r1, r2, respectively.
bool circle_tangent(PT a, double r1, PT b, double r2, double R, pair<PT, PT>& posible_circles) {
    if (abs(a - b) - r1 - r2 - 2*R < eps) return false;
    double v = R + r2;
    double u = R + r1;
    double d = abs(a - b);
    double theta = acos((u*u + d*d - v*v) / (2*u*d));
    PT p1 = rotate_ccw(b, a, theta);
    PT p2 = rotate_ccw(b, a, -theta);
    p1 = p1 * (u / abs(p1 - a));
    p2 = p2 * (u / abs(p2 - a));
    posible_circles = MP(p1, p2);
    return true;
}

// The Great-Circle Distance between any two points A and B on sphere
// is the shortest distance along a path on the surface of the sphere
double greater_circle_distance(double pLat, double pLong, double qLat, double qLong, double radius) {
    pLat *= PI / 180; pLong *= PI / 180; // conversion from degree to radian
    qLat *= PI / 180; qLong *= PI / 180;
    return radius * acos(cos(pLat)*cos(pLong)*cos(qLat)*cos(qLong) +
            cos(pLat)*sin(pLong)*cos(qLat)*sin(qLong) +
            sin(pLat)*sin(qLat));
}

double circle_area(double R) {
    return PI * R*R;
}

// given circle centered at 0,0 with radius R, finds the area that lies to the
// right of the line x = d
double circle_segment_area(double R, double d) {
    if (d == 0) return circle_area(R) / 2;
    double hh = (R*R - d*d);
    double theta = 2 * acos((R*R + d*d - hh) / (2*R*d));
    return R*R * (theta - sin(theta)) / 2;
}

// return volume of cap sphere with radius R, cap height h
double spherical_cap_volume(double R, double h) {
    return PI * h * h * (3.0 * R - h) / 3.0;
}

double spherical_cap_area(double R, double h) {
    return 2 * PI * R * h;
}

int main() {
    // circle c(-2, 5, sqrt(10)); //(x+2)^2+(y-5)^2=10
    // assert(c == circle(PT(-2, 5), sqrt(10)));
    // assert(c == circle(PT(1, 6), PT(-5, 4)));
    // assert(c == circle(PT(-3, 2), PT(-3, 8), PT(-1, 8)));
    // assert(c == incircle(PT(-12, 5), PT(3, 0), PT(0, 9)));
    // assert(c.contains(PT(-2, 8)) && !c.contains(PT(-2, 9)));
    // assert(c.on_edge(PT(-1, 2)) && !c.on_edge(PT(-1.01, 2)));

    // pair<PT, PT> ct = circle_tangent(PT(0,0), 6, PT(10, 0), 6, 10);
    // cout << ct.first << "\n" << ct.second << "\n";

    circle c1 = MP(PT(0,0), 10);
    circle c2 = MP(PT(30,0), 5);
    pair<Segment,Segment> t;
    if (inner_tangent(c1, c2, t)) {
        cout << t.A.A << " " << t.A.B << "\n";
        cout << t.B.A << " " << t.B.B << "\n";
    }

    return 0;
}
```

4.2.2 Minimum Enclosing Circle

```
/*

   Given a range of points on the 2D cartesian plane, determine
   the equation of the circle with smallest possible area which
   encloses all of the points. Note: in an attempt to avoid the
   worst case, the circles are randomly shuffled before the
   algorithm is performed. This is not necessary to obtain the
   correct answer, and may be removed if the input order must
   be preserved.

   Time Complexity: O(n) average on the number of points given.

REQUIRES:
line_line
circle_center (for both three points and diameter made from two points)
 */

template<class It> circle smallest_circle(It lo, It hi) {
    if (lo == hi) return circle_center(0, 0, 0);
    if (lo + 1 == hi) return circle_center(*lo, 0);
    random_shuffle(lo, hi);
    circle res = circle_center(*lo, *(lo + 1));
    for (It i = lo + 2; i != hi; ++i) {
        if (contains(res, *i)) continue;
        res = circle_center(*lo, *i);
        for (It j = lo + 1; j != i; ++j) {
            if (contains(res, *j)) continue;
            res = circle_center(*i, *j);
            for (It k = lo; k != j; ++k)
                if (!contains(res, *k)) res = circle_center(*i, *j, *k);
        }
    }
    return res;
}

int main() {
    vector<PT> v; // some points
    circle ans = smallest_circle(v.begin(), v.end());
}
```

4.3 Fundamentals

4.3.1 Template

```
#include <bits/stdc++.h>
using namespace std;
typedef complex<double> PT;

const double eps = 1e-9;
#define A first
#define B second
#define X real()
#define Y imag()
```

```
const double PI = acos(-1);
#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LT(a, b) ((a) < (b) - eps)         /* less than */
#define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */
#define NE(a, b) (fabs((a) - (b)) > eps)  /* not equal to */

double dot(const PT& p, const PT& q) { return real(conj(p) * q); }
double cross(const PT& p, const PT& q) { return imag(conj(p) * q); }

// rotate PT a around PT o by th radians
PT rotate_ccw(const PT& a, const PT& o, double th) {
    return (a-o) * polar(1.0, th) + o;
}

//  does p -> q -> r rotates CCW
bool is_ccw(const PT& p, const PT& q, const PT& r) {
    return cross(q - p, r - p) > eps;
}

// angle to a, around b, from c, CCW + CW -
double angle(const PT& a, const PT& b, const PT& c) {
    return remainder(arg(a-b) - arg(c-b), 2.0 * M_PI);
}

//  are a, b, c collinear
bool is_collinear(const PT& a, const PT& b, const PT& c) {
    return abs(cross(b - a, c - a)) < eps;
}

//  whether lines a-b and c-d are parallel (or collinear)
bool is_parallel(const PT& a, const PT& b, const PT& c, const PT& d) {
    return abs(cross(b - a, c - d)) < eps;
}

//  intersection of lines a1-a2 and b1-b2
bool line_line(PT a1, PT a2, PT b1, PT b2, PT &isect) {
    double d = cross(a2-a1, b2-b1);
    if (abs(d) < eps) return false;
    double t = cross(b2-b1, a1-b1) / d;
    isect = a1 + (a2-a1)*t;
    return true;
}

// project PT p on line (0,0) - v
PT project(const PT& p, const PT& v) {
    return v * dot(p, v) / norm(v);
}

// project PT p onto line a, b
PT project(const PT& p, const PT& a, const PT& b) {
    return a + (b - a) * dot(p - a, b - a) / norm(b - a);
}

//  nearest PT to p, on segment a-b
PT nearest_on_segment(const PT& p, const PT& a, const PT& b) {
    double r = dot(b - a, b - a);
    if (abs(r) < eps) return a;
    r = dot(p - a, b - a) / r;
    if (r < 0) return a;
    if (r > 1) return b;
    return a + (b - a)*r;
}

//  whether p is on segment a-b
bool is_on_segment(const PT& p, const PT& a, const PT& b) {
    return abs(abs(a - p) + abs(b - p) - abs(a - b)) < eps;
}

//  reflect p around line a-b
PT reflect(const PT& p, const PT& a, const PT& b) {
    PT z = p - a;
    PT w = b - a;
    return conj(z / w) * w + a;
}

//  whether PT a is on boundary of simple polygon p
bool is_on_polygon(const PT& a, const vector<PT> &p) {
    for (int i = 0; i < (int)p.size(); i++)
        if (is_on_segment(a, p[i], p[(i + 1) % (int)p.size()]))
            return true;
    return false;
}

//  whether PT a is inside a simple polygon p
bool is_in_polygon(const PT& a, const vector<PT> &p) {
    double sum = 0;
    for (int i = 0; i < (int)p.size(); i++) {
        sum += angle(p[i], a, p[(i + 1) % (int)p.size()]);
    }
    return abs(abs(sum) - 2*M_PI) < eps;
}

//  area of a simple polygon p
double area_polygon(const vector<PT> &p) {
    double area = 0;
    for (int i = 0; i < (int)p.size(); i++)
        area += cross(p[i], p[(i + 1) % (int)p.size()]);
    return abs(area) / 2.0;                                             //  CCW + CW -
}

//  area of triangle using Heron's formula
double area_heron(double x, double y, double z) {
    double s = 0.5 * (x + y + z);
    return sqrt(s) * sqrt(s - x) * sqrt(s - y) * sqrt(s - z);
}

//  radius of the (inner and outer) fourth circle in a set tangent at six distinct PTs using Descartes's theorem
pair<double, double> kissing_circle(double r1, double r2, double r3) {
    double k1 = 1 / r1, k2 = 1 / r2, k3 = 1 / r3;
    double s1 = k1 + k2 + k3;
    double s2 = 2 * sqrt(k1 * k2 + k2 * k3 + k3 * k1);
    return make_pair(1/(s1 + s2), 1/(s1 - s2));         //  2nd may be negative showing including circle
}

template<class It> PT centroid(It lo, It hi) {
    if (lo == hi) return PT(0, 0);
    double xtot = 0, ytot = 0, cnt = hi - lo;
    for (; lo != hi; ++lo) {
        xtot += lo->X;
        ytot += lo->Y;
    }
    return PT(xtot / cnt, ytot / cnt);
```

```
}
```

4.4 Lines

4.4.1 Lines

```
/*
   A 2D line is expressed in the form Ax + By + C = 0.
   Given the line form, we can get two points on the line.
 */

struct line {
    double a, b, c;
    line(): a(0), b(0), c(0) {} //invalid or uninitialized line
    line(double A, double B, double C) : a(A), b(B), c(C) {}

    //solve for y, given x
    //for vertical lines, either +inf, -inf, or nan is returned
    double y(const double& x) const {
        return (-c - a * x) / b;
    }

    pair<PT, PT> get_points() {
        PT p1, p2;
        if (EQ(b, 0)) { // vertical line
            p1 = PT(-c/a, 0);
            p2 = PT(-c/a, 10);
        } else {
            p1 = PT(0, y(0));
            p2 = PT(10, y(10));
        }
        return MP(p1, p2);
    }
};


int main() {

    // random
    line l1 = line(1,2,3);
    pair<PT, PT> p = l1.get_points();
    cerr << p.first << " " << p.second << "\n";

    // vertical
    line l2 = line(1,0,3);
    p = l2.get_points();
    cerr << p.first << " " << p.second << "\n";

    // horizontal
    line l3 = line(0,2,3);
    p = l3.get_points();
    cerr << p.first << " " << p.second << "\n";

    return 0;
}
```

4.4.2 Segment Intersection

```
/*

   Given a range of segments on the Cartesian plane, identify one
   pair of segments which intersect each other. This is done using
   a sweep line algorithm.

   Time Complexity: O(n log n) where n is the number of segments.

 */

#include <algorithm> /* std::min(), std::max(), std::sort() */
#include <cmath>     /* fabs() */
#include <set>
#include <utility>   /* std::pair */

typedef std::pair<double, double> point;
#define x first
#define y second

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps)  /* equal to */
#define LT(a, b) ((a) < (b) - eps)         /* less than */
#define LE(a, b) ((a) <= (b) + eps)        /* less than or equal to */

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }
double cross(const point & o, const point & a, const point & b) {
    return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

const bool TOUCH_IS_INTERSECT = true;

bool contain(const double & l, const double & m, const double & h) {
    if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
    return LT(l, m) && LT(m, h);
}

bool overlap(const double & l1, const double & h1,
        const double & l2, const double & h2) {
    if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
    return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
        const point & c, const point & d) {
    point ab(b.x - a.x, b.y - a.y);
    point ac(c.x - a.x, c.y - a.y);
    point cd(d.x - c.x, d.y - c.y);
    double c1 = cross(ab, cd), c2 = cross(ac, ab);
    if (EQ(c1, 0) && EQ(c2, 0)) {
        double t0 = dot(ac, ab) / norm(ab);
        double t1 = t0 + dot(cd, ab) / norm(ab);
        if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
            point res1 = std::max(std::min(a, b), std::min(c, d));
            point res2 = std::min(std::max(a, b), std::max(c, d));
            return (res1 == res2) ? 0 : 1;
        }
        return -1;
    }
    if (EQ(c1, 0)) return -1;
    double t = cross(ac, cd) / c1, u = c2 / c1;
    if (contain(0, t, 1) && contain(0, u, 1)) return 0;
    return -1;
}

struct segment {
    point p, q;

    segment() {}
    segment(const point & p, const point & q) {
        if (p < q) {
            this->p = p;
            this->q = q;
        } else {
            this->p = q;
            this->q = p;
        }
    }

    bool operator < (const segment & rhs) const {
        if (p.x < rhs.p.x) {
            double c = cross(p, q, rhs.p);
            if (c != 0) return c > 0;
        } else if (p.x > rhs.p.x) {
            double c = cross(rhs.p, rhs.q, q);
            if (c != 0) return c < 0;
        }
        return p.y < rhs.p.y;
    }
};

template<class SegIt> struct event {
    point p;
    int type;
    SegIt seg;

    event() {}
    event(const point & p, const int type, SegIt seg) {
        this->p = p;
        this->type = type;
        this->seg = seg;
    }

    bool operator < (const event & rhs) const {
        if (p.x != rhs.p.x) return p.x < rhs.p.x;
        if (type != rhs.type) return type > rhs.type;
        return p.y < rhs.p.y;
    }
};

bool intersect(const segment & s1, const segment & s2) {
    return seg_intersection(s1.p, s1.q, s2.p, s2.q) >= 0;
}

//returns whether any pair of segments in the range [lo, hi) intersect
//if the result is true, one such intersection pair will be stored
//into values pointed to by res1 and res2.
template<class It>
bool find_intersection(It lo, It hi, segment * res1, segment * res2) {
    int cnt = 0;
    event<It> e[2 * (hi - lo)];
    for (It it = lo; it != hi; ++it) {
        if (it->p > it->q) std::swap(it->p, it->q);
        e[cnt++] = event<It>(it->p, 1, it);
        e[cnt++] = event<It>(it->q, -1, it);
    }
    std::sort(e, e + cnt);
    std::set<segment> s;
    std::set<segment>::iterator it, next, prev;
    for (int i = 0; i < cnt; i++) {
        It seg = e[i].seg;
        if (e[i].type == 1) {
            it = s.lower_bound(*seg);
            if (it != s.end() && intersect(*it, *seg)) {
                *res1 = *it; *res2 = *seg;
                return true;
            }
            if (it != s.begin() && intersect(*--it, *seg)) {
                *res1 = *it; *res2 = *seg;
                return true;
            }
            s.insert(*seg);
        } else {
            it = s.lower_bound(*seg);
            next = prev = it;
            prev = it;
            if (it != s.begin() && it != --s.end()) {
                ++next;
                --prev;
                if (intersect(*next, *prev)) {
                    *res1 = *next; *res2 = *prev;
                    return true;
                }
            }
            s.erase(it);
        }
    }
    return false;
}

/*** Example Usage ***/

#include <iostream>
#include <vector>
using namespace std;

void print(const segment & s) {
    cout << "(" << s.p.x << "," << s.p.y << ")<->";
    cout << "(" << s.q.x << "," << s.q.y << ")\n";
}

int main() {
    vector<segment> v;
    v.push_back(segment(point(0, 0), point(2, 2)));
    v.push_back(segment(point(3, 0), point(0, -1)));
    v.push_back(segment(point(0, 2), point(2, -2)));
    v.push_back(segment(point(0, 3), point(9, 0)));
    segment res1, res2;
    bool res = find_intersection(v.begin(), v.end(), &res1, &res2);
    if (res) {
        print(res1);
        print(res2);
    } else {
        cout << "No intersections.\n";
    }
    return 0;
}
```

4.5 Points

4.5.1 Closest Pair

```
/*

   Given a range containing distinct PTs on the Cartesian plane,
   determine two PTs which have the closest possible distance.
   A divide and conquer algorithm is used. Note that the ordering
   of PTs in the input range may be changed by the function.

   Time Complexity: O(n log^2 n) where n is the number of PTs.

 */

bool cmp_x(const PT& a, const PT& b) { return a.X < b.X; }
bool cmp_y(const PT& a, const PT& b) { return a.Y < b.Y; }

template<class It>
double rec(It lo, It hi, pair<PT, PT>& res, double mindist) {
    if (lo == hi) return DBL_MAX;
    It mid = lo + (hi - lo) / 2;
    double midx = mid->X;
    double d1 = rec(lo, mid, res, mindist);
    mindist = min(mindist, d1);
    double d2 = rec(mid + 1, hi, res, mindist);
    mindist = min(mindist, d2);
    sort(lo, hi, cmp_y);
    int size = 0;
    It t[hi - lo];
    for (It it = lo; it != hi; ++it)
        if (fabs(it->X - midx) < mindist)
            t[size++] = it;
    FOR(i,0,size) {
        FOR(j,i+1,size) {
            PT a = *t[i], b = *t[j];
            if (b.Y - a.Y >= mindist) break;
            double dist = norm(a - b);
            if (mindist > dist) {
                mindist = dist;
                res = MP(a, b);
            }
        }
    }
    return mindist;
}

template<class It> pair<PT, PT> closest_pair(It lo, It hi) {
    pair<PT, PT> res;
    sort(lo, hi, cmp_x);
    rec(lo, hi, res, DBL_MAX);
    return res;
}

int main() {
    vector<PT> v; // some points
    pair<PT, PT> sol = closest_pair(v.begin(), v.end());
    return 0;
}
```

4.5.2 Delaunay

```
/*

   Given a range of points P on the Cartesian plane, the Delaunay
   Triangulation of said points is a set of non-overlapping triangles
   covering the entire convex hull of P, such that no point in P lies
   within the circumcircle of any of the resulting triangles. The
   triangulation maximizes the minimum angle of all the angles of the
   triangles in the triangulation. In addition, for any point p in the
   convex hull (not necessarily in P), the nearest point is guaranteed
   to be a vertex of the enclosing triangle from the triangulation.
See: https://en.wikipedia.org/wiki/Delaunay_triangulation

The triangulation may not exist (e.g. for a set of collinear points)
or it may not be unique (multiple possible triangulations may exist).
The triangulation may not exist (e.g. for a set of collinear points)
or it may not be unique (multiple possible triangulations may exist).
The following program assumes that a triangulation exists, and
produces one such valid result using one of the simplest algorithms
to solve this problem. It involves encasing the simplex in a circle
and rejecting the simplex if another point in the tessellation is
within the generalized circle.

Time Complexity: O(n^4) on the number of input points.

 */

#include <algorithm> /* std::sort() */
#include <cmath>     /* fabs(), sqrt() */
#include <utility>   /* std::pair */
#include <vector>

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define GT(a, b) ((a) > (b) + eps)        /* greater than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */
#define GE(a, b) ((a) >= (b) - eps)       /* greater than or equal to */

typedef std::pair<double, double> point;
#define x first
#define y second

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

const bool TOUCH_IS_INTERSECT = false;

bool contain(const double & l, const double & m, const double & h) {
    if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
    return LT(l, m) && LT(m, h);
}
```

```
bool overlap(const double & l1, const double & h1,
        const double & l2, const double & h2) {
    if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
    return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
        const point & c, const point & d) {
    point ab(b.x - a.x, b.y - a.y);
    point ac(c.x - a.x, c.y - a.y);
    point cd(d.x - c.x, d.y - c.y);
    double c1 = cross(ab, cd), c2 = cross(ac, ab);
    if (EQ(c1, 0) && EQ(c2, 0)) {
        double t0 = dot(ac, ab) / norm(ab);
        double t1 = t0 + dot(cd, ab) / norm(ab);
        if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
            point res1 = std::max(std::min(a, b), std::min(c, d));
            point res2 = std::min(std::max(a, b), std::max(c, d));
            return (res1 == res2) ? 0 : 1;
        }
        return -1;
    }
    if (EQ(c1, 0)) return -1;
    double t = cross(ac, cd) / c1, u = c2 / c1;
    if (contain(0, t, 1) && contain(0, u, 1)) return 0;
    return -1;
}

struct triangle { point a, b, c; };

template<class It>
std::vector<triangle> delaunay_triangulation(It lo, It hi) {
    int n = hi - lo;
    std::vector<double> x, y, z;
    for (It it = lo; it != hi; ++it) {
        x.push_back(it->x);
        y.push_back(it->y);
        z.push_back((it->x) * (it->x) + (it->y) * (it->y));
    }
    std::vector<triangle> res;
    for (int i = 0; i < n - 2; i++) {
        for (int j = i + 1; j < n; j++) {
            for (int k = i + 1; k < n; k++) {
                if (j == k) continue;
                double nx = (y[j] - y[i]) * (z[k] - z[i]) - (y[k] - y[i]) * (z[j] - z[i]);
                double ny = (x[k] - x[i]) * (z[j] - z[i]) - (x[j] - x[i]) * (z[k] - z[i]);
                double nz = (x[j] - x[i]) * (y[k] - y[i]) - (x[k] - x[i]) * (y[j] - y[i]);
                if (GE(nz, 0)) continue;
                bool done = false;
                for (int m = 0; m < n; m++)
                    if (x[m] - x[i]) * nx + (y[m] - y[i]) * ny + (z[m] - z[i]) * nz > 0) {
                        done = true;
                        break;
                    }
                if (!done) { //handle 4 points on a circle
                    point s1[] = { *(lo + i), *(lo + j), *(lo + k), *(lo + i) };
                    for (int t = 0; t < (int)res.size(); t++) {
                        point s2[] = { res[t].a, res[t].b, res[t].c, res[t].a };
                        for (int u = 0; u < 3; u++)
                            for (int v = 0; v < 3; v++)
                                if (seg_intersection(s1[u], s1[u + 1], s2[v], s2[v + 1]) == 0)
                                    goto skip;
                    }
                    res.push_back((triangle){*(lo + i), *(lo + j), *(lo + k)});
                }
skip:;
            }
        }
    }
    return res;
}

/*** Example Usage ***/

#include <iostream>
using namespace std;

int main() {
    vector<point> v;
    v.push_back(point(1, 3));
    v.push_back(point(1, 2));
    v.push_back(point(2, 1));
    v.push_back(point(0, 0));
    v.push_back(point(-1, 3));
    vector<triangle> dt = delaunay_triangulation(v.begin(), v.end());
    for (int i = 0; i < (int)dt.size(); i++) {
        cout << "Triangle: ";
        cout << "(" << dt[i].a.x << "," << dt[i].a.y << ") ";
        cout << "(" << dt[i].b.x << "," << dt[i].b.y << ") ";
        cout << "(" << dt[i].c.x << "," << dt[i].c.y << ")\n";
    }
    return 0;
}
```

4.5.3 Diameter

```
/*

   Determines the diametral pair of a range of PTs. The diamter
   of a set of PTs is the largest distance between any two
   PTs in the set. A diametral pair is a pair of PTs in the
   set whose distance is equal to the set's diameter. The following
   program uses rotating calipers method to find a solution.

   Time Complexity: O(n log n) on the number of PTs in the set.

 */

double area(const PT & o, const PT & a, const PT & b) {
    return fabs(cross(o, a, b));
}

template<class It> pair<PT, PT> diametral_pair(It lo, It hi) {
    vector<PT> h = convex_hull(lo, hi);
    int m = h.size();
    if (m == 1) return make_pair(h[0], h[0]);
    if (m == 2) return make_pair(h[0], h[1]);
    int k = 1;
    while (area(h[m - 1], h[0], h[(k + 1) % m]) > area(h[m - 1], h[0], h[k]))
        k++;
    double maxdist = 0, d;
    pair<PT, PT> res;
    for (int i = 0, j = k; i <= k && j < m; i++) {
        d = norm(h[i] - h[j]);
        if (d > maxdist) {
            maxdist = d;
            res = make_pair(h[i], h[j]);
        }
        while (j < m && area(h[i], h[(i + 1) % m], h[(j + 1) % m]) >
                area(h[i], h[(i + 1) % m], h[j])) {
            d = norm(h[i] - h[(j + 1) % m]);
            if (d > maxdist) {
                maxdist = d;
                res = make_pair(h[i], h[(j + 1) % m]);
            }
            j++;
        }
    }
    return res;
}

/*** Example Usage ***/

int main() {
    vector<PT> v;
    v.push_back(PT(0, 0));
    v.push_back(PT(3, 0));
    v.push_back(PT(0, 3));
    v.push_back(PT(1, 1));
    v.push_back(PT(4, 4));
    pair<PT, PT> res = diametral_pair(v.begin(), v.end());
    cout << "diametral pair: (" << res.first.X << "," << res.first.Y << ") ";
    cout << "(" << res.second.X << "," << res.second.Y << ")\n";
    cout << "diameter: " << abs(res.first - res.second) << "\n";
    return 0;
}
```

4.6 Polygons

4.6.1 Convex Cut

```
/*
   Given a range of PTs specifying a polygon on the Cartesian
   plane, as well as two PTs specifying an infinite line, "cut"
   off the right part of the polygon with the line and return the
   resulting polygon that is the left part.
   Time Complexity: O(n) on the number of PTs in the poylgon.

NEED: line_intersection from Lines section
 */

int orientation(const PT & o, const PT & a, const PT & b) {
    double c = cross(a - o, b - o);
    return LT(c, 0) ? -1 : (GT(c, 0) ? 1 : 0);
}

template<class It>
vector<PT> convex_cut(It lo, It hi, const PT & p, const PT & q) {
    vector<PT> res;
    for (It i = lo, j = hi - 1; i != hi; j = i++) {
        int d1 = orientation(p, q, *j), d2 = orientation(p, q, *i);
        if (d1 >= 0) res.push_back(*j);
        if (d1 * d2 < 0) {
            PT r;
            line_intersection(p, q, *j, *i, &r);
            res.push_back(r);
        }
    }
    return res;
}

int main() {
    vector<PT> polygon;
    vector<PT> cut = convex_cut(polygon.begin(), polygon.end(),
            PT(100, h), PT(-100, h)); // some points determining a line
    return 0;
}
```

4.6.2 Convex Hull

```
/*
   Determines the convex hull from a range of PTs, that is, the
   smallest convex polygon (a polygon such that every line which
   crosses through it will only cross through it once) that contains
   all of the PTs. This function uses the monotone chain algorithm
   to compute the uperp and lower hulls separately.
Returns: a vector of the convex hull PTs in clockwise order.
Complexity: O(n log n) on the number of PTs given
Notes: To yield the hull PTs in counterclockwise order,
replace every  usageof GE() in the function with LE().
To have the first PT on the hull repeated as the last,
replace the last line of the function to res.resize(k);
 */

//change < 0 comparisons to > 0 to produce hull PTs in CCW order
double cw(const PT & o, const PT & a, const PT & b) {
    return cross(a - o, b - o) < -eps;
}

//convex hull from a range [lo, hi) of PTs
//monotone chain in O(n log n) to find hull PTs in CW order
//notes: the range of input PTs will be sorted lexicographically
template<class It> vector<PT> convex_hull(It lo, It hi) {
    int k = 0;
    if (hi - lo <= 1) return vector<PT>(lo, hi);
    vector<PT> res(2 * (int)(hi - lo));
    sort(lo, hi, [](PT a, PT b) {
            return MP(a.X, a.Y) < MP(b.X, b.Y);
            });
    for (It it = lo; it != hi; ++it) {
        while (k >= 2 && !cw(res[k - 2], res[k - 1], *it)) k--;
        res[k++] = *it;
    }
    int t = k + 1;
    for (It it = hi - 2; it != lo - 1; --it) {
        while (k >= t && !cw(res[k - 2], res[k - 1], *it)) k--;
        res[k++] = *it;
    }
    res.resize(k - 1);
    return res;
}
```

```
int main() {
    vector<PT> v; // init points.
    vector<PT> hull = convex_hull(v.begin(), v.end());
}
```

4.6.3 Polygon Union Area

```
/*
   given any polygons, find the area of their union.
 */
inline double area(double a, double b, const vector<vector<PT> >& v) {
    vector<pair<double, int> > event;
    int cnt = 0;
    FOR(i,0,SZ(v)) {
        FOR(k,0,SZ(v[i])) {
            PT p1 = v[i][k];
            PT p2 = v[i][(k + 1) % SZ(v[i])];
            if (p1.X > p2.X) swap(p1, p2);
            if (p1.X <= a && p2.X >= b) {
                PT q1, q2;
                lineline(p1, p2, PT(a,0), PT(a,100), q1);
                lineline(p1, p2, PT(b,0), PT(b,100), q2);
                double yy = ((q1.Y + q2.Y) / 2);
                event.push_back((MP(yy, i)));
            }
        }
    }
    sort(event.begin(), event.end());
    double ans = 0, amnt = 0;
    set<int> met;
    FOR(i,0,SZ(event)) {
        double yy = event[i].A;
        int t = event[i].B;
        if (met.count(t)) {
            amnt --;
            if (amnt == 0)
                ans += yy;
            met.erase(t);
        } else {
            if (amnt == 0)
                ans -= yy;
            amnt ++;
            met.insert(t);
        }
    }
    return ans * (b - a);
}

double polygon_union_area(const vector<vector<PT> >& v) {
    int N = SZ(v);
    vector<double> xs;
    vector<pair<PT, PT> > all;
    FOR(i,0,N) {
        FOR(k,0,SZ(v[i])) {
            int j = (k + 1) % SZ(v[i]);
            all.push_back(MP(v[i][k], v[i][j]));
        }
    }
    FOR(i,0,SZ(all)) {
        xs.push_back(all[i].A.X);
        FOR(k,0,i) {
            PT p;
            if (lineline(all[i].A, all[i].B, all[k].A, all[k].B, p)) {
                if (p.X >= 0 && p.X <= 1000 && p.Y <= 1000 && p.Y >= 0)
                    xs.push_back(p.X);
            }
        }
    }
    double ans = 0;
    sort(xs.begin(), xs.end());
    for (int i = 0; i < xs.size() - 1; i++) {
        if (xs[i + 1] - xs[i] < eps) continue;
        ans += area(xs[i], xs[i + 1], v);
    }
    return ans;
}
```

4.6.4 Polygon Union Intersection

```
/*

   Given two ranges of points respectively denoting the vertices of
   two polygons, determine the intersection area of those polygons.
   Using this, we can easily calculate their union with the forumla:
   union_area(A, B) = area(A) + area(B) - intersection_area(A, B)

   Time Complexity: O(n^2 log n), where n is the total number of vertices.

 */

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

const double eps = 1e-9;

#define EQ(a, b) (fabs((a) - (b)) <= eps) /* equal to */
#define LT(a, b) ((a) < (b) - eps)        /* less than */
#define LE(a, b) ((a) <= (b) + eps)       /* less than or equal to */

typedef std::pair<double, double> point;
#define x first
#define y second

inline int sgn(const double & x) {
    return (0.0 < x) - (x < 0.0);
}

//Line and line segment intersection (see their own sections)

int line_intersection(const point & p1, const point & p2,
        const point & p3, const point & p4, point * p = 0) {
    double a1 = p2.y - p1.y, b1 = p1.x - p2.x;
    double c1 = -(p1.x * p2.y - p2.x * p1.y);
    double a2 = p4.y - p3.y, b2 = p3.x - p4.x;
    double c2 = -(p3.x * p4.y - p4.x * p3.y);
    double x = -(c1 * b2 - c2 * b1), y = -(a1 * c2 - a2 * c1);
    double det = a1 * b2 - a2 * b1;
    if (EQ(det, 0))
        return (EQ(x, 0) && EQ(y, 0)) ? 1 : -1;
    if (p != 0) *p = point(x / det, y / det);
    return 0;
}

double norm(const point & a) { return a.x * a.x + a.y * a.y; }
double dot(const point & a, const point & b) { return a.x * b.x + a.y * b.y; }
double cross(const point & a, const point & b) { return a.x * b.y - a.y * b.x; }

const bool TOUCH_IS_INTERSECT = true;

bool contain(const double & l, const double & m, const double & h) {
    if (TOUCH_IS_INTERSECT) return LE(l, m) && LE(m, h);
    return LT(l, m) && LT(m, h);
}

bool overlap(const double & l1, const double & h1,
        const double & l2, const double & h2) {
    if (TOUCH_IS_INTERSECT) return LE(l1, h2) && LE(l2, h1);
    return LT(l1, h2) && LT(l2, h1);
}

int seg_intersection(const point & a, const point & b,
        const point & c, const point & d,
        point * p = 0, point * q = 0) {
    point ab(b.x - a.x, b.y - a.y);
    point ac(c.x - a.x, c.y - a.y);
    point cd(d.x - c.x, d.y - c.y);
    double c1 = cross(ab, cd), c2 = cross(ac, ab);
    if (EQ(c1, 0) && EQ(c2, 0)) { //collinear
        double t0 = dot(ac, ab) / norm(ab);
        double t1 = t0 + dot(cd, ab) / norm(ab);
        if (overlap(std::min(t0, t1), std::max(t0, t1), 0, 1)) {
            point res1 = std::max(std::min(a, b), std::min(c, d));
            point res2 = std::min(std::max(a, b), std::max(c, d));
            if (res1 == res2) {
                if (p != 0) *p = res1;
                return 0; //collinear, meeting at an endpoint
            }
            if (p != 0 && q != 0) *p = res1, *q = res2;
            return 1; //collinear and overlapping
        } else {
            return -1; //collinear and disjoint
        }
    }
    if (EQ(c1, 0)) return -1; //parallel and disjoint
    double t = cross(ac, cd) / c1, u = c2 / c1;
    if (contain(0, t, 1) && contain(0, u, 1)) {
        if (p != 0) *p = point(a.x + t * ab.x, a.y + t * ab.y);
        return 0; //non-parallel with one intersection
    }
    return -1; //non-parallel with no intersections
}

struct event {
    double y;
    int mask_delta;

    event(double y = 0, int mask_delta = 0) {
        this->y = y;
        this->mask_delta = mask_delta;
    }

    bool operator < (const event & e) const {
        if (y != e.y) return y < e.y;
        return mask_delta < e.mask_delta;
    }
};

template<class It>
double intersection_area(It lo1, It hi1, It lo2, It hi2) {
    It plo[2] = {lo1, lo2}, phi[] = {hi1, hi2};
    std::set<double> xs;
    for (It i1 = lo1; i1 != hi1; ++i1) xs.insert(i1->x);
    for (It i2 = lo2; i2 != hi2; ++i2) xs.insert(i2->x);
    for (It i1 = lo1, j1 = hi1 - 1; i1 != hi1; j1 = i1++) {
        for (It i2 = lo2, j2 = hi2 - 1; i2 != hi2; j2 = i2++) {
            point p;
            if (seg_intersection(*i1, *j1, *i2, *j2, &p) == 0)
                xs.insert(p.x);
        }
    }
    std::vector<double> xsa(xs.begin(), xs.end());
    double res = 0;
    for (int k = 0; k < (int)xsa.size() - 1; k++) {
        double x = (xsa[k] + xsa[k + 1]) / 2;
        point sweep0(x, 0), sweep1(x, 1);
        std::vector<event> events;
        for (int poly = 0; poly < 2; poly++) {
            It lo = plo[poly], hi = phi[poly];
            double area = 0;
            for (It i = lo, j = hi - 1; i != hi; j = i++)
                area += (j->x - i->x) * (j->y + i->y);
            for (It j = lo, i = hi - 1; j != hi; i = j++) {
                point p;
                if (line_intersection(*j, *i, sweep0, sweep1, &p) == 0) {
                    double y = p.y, x0 = i->x, x1 = j->x;
                    if (x0 < x && x1 > x) {
                        events.push_back(event(y,  sgn(area) * (1 << poly)));
                    } else if (x0 > x && x1 < x) {
                        events.push_back(event(y, -sgn(area) * (1 << poly)));
                    }
                }
            }
        }
        std::sort(events.begin(), events.end());
        double a = 0.0;
        int mask = 0;
        for (int j = 0; j < (int)events.size(); j++) {
            if (mask == 3)
                a += events[j].y - events[j - 1].y;
            mask += events[j].mask_delta;
        }
        res += a * (xsa[k + 1] - xsa[k]);
    }
    return res;
}

template<class It> double polygon_area(It lo, It hi) {
    if (lo == hi) return 0;
    double area = 0;
    if (*lo != *--hi)
        area += (lo->x - hi->x) * (lo->y + hi->y);
    for (It i = hi, j = hi - 1; i != lo; --i, --j)
        area += (i->x - j->x) * (i->y + j->y);
    return fabs(area / 2.0);
```

```cpp
}

template<class It>
double union_area(It lo1, It hi1, It lo2, It hi2) {
    return polygon_area(lo1, hi1) + polygon_area(lo2, hi2) -
        intersection_area(lo1, hi1, lo2, hi2);
}

int main() {
    vector<point> p1, p2;

    //irregular pentagon with area 1.5 triangle in quadrant 2
    p1.push_back(point(1, 3));
    p1.push_back(point(1, 2));
    p1.push_back(point(2, 1));
    p1.push_back(point(0, 0));
    p1.push_back(point(-1, 3));
    //a big square in quadrant 2
    p2.push_back(point(0, 0));
    p2.push_back(point(0, 3));
    p2.push_back(point(-3, 3));
    p2.push_back(point(-3, 0));

    assert(EQ(1.5, intersection_area(p1.begin(), p1.end(),
                    p2.begin(), p2.end())));
    assert(EQ(12.5, union_area(p1.begin(), p1.end(),
                    p2.begin(), p2.end())));
    return 0;
}
```

5 Graphs and Trees

5.1 Auxiliary Tree

5.1.1 Auxiliary Tree

```cpp
// Kingdom and Cities - Codeforces.
// Solution: Auxiliary Tree

#include "bits/stdc++.h"
using namespace std;

const int N   = 1e5 + 5;
const int LN  = 18;
const int INF = 1e8 + 8;

int n, q, cur_time, len, ans;
int tin[N], tout[N], depth[N], parent[N], val[N], dp[LN][N];
bool important[N];
vector < int > adj[N], aux[N];
vector < int > nodes;

inline void dfs_prep(int u, int p){
    tin[u] = ++cur_time;
    dp[0][u] = parent[u] = p;
    for(int i = 1; i < LN; i++) dp[i][u] = dp[i - 1][dp[i - 1][u]];
    for(int v : adj[u]){
        if(v != p){
            depth[v] = depth[u] + 1;
            dfs_prep(v, u);
        }
    }
    tout[u] = cur_time;
}

inline int lca(int u, int v){
    if(depth[u] < depth[v]) swap(u, v);
    for(int i = LN - 1; i >= 0; i--){
        if(depth[u] - (1 << i) >= depth[v])
            u = dp[i][u];
    }
    if(u == v) return u;
    for(int i = LN - 1; i >= 0; i--){
        if(dp[i][u] != dp[i][v])
            u = dp[i][u], v = dp[i][v];
    }
    return parent[u];
}

inline bool compare(int u, int v){
    return (tin[u] < tin[v]);
}

inline void clean(vector < int > &x){
    sort(x.begin(), x.end(), compare);
    x.resize(unique(x.begin(), x.end()) - x.begin());
    len = (int) nodes.size();
}

inline void dfs(int u){
    val[u] = INF;
    int min_val = INF, noob_child = 0;
    for(int v : aux[u]){
        dfs(v);
        if(val[v] != INF) ++noob_child;
        min_val = min(min_val, val[v]);
    }
    if(!important[u]){
        if(noob_child > 1) ans += 1;
        else val[u] = min_val;
    }
    else{
        val[u] = 1;
        ans += noob_child;
    }
}

inline bool is_ancestor(int u, int v){
    return ((tin[v] >= tin[u]) && (tin[v] <= tout[u]));
}

inline void solve(bool rekt){
    if(rekt){
        printf("-1\n");
        return;
    }
    clean(nodes);
    for(int i = 0; i < len - 1; i++){
        int lc = lca(nodes[i], nodes[i + 1]);
        nodes.push_back(lc);
    }
    clean(nodes);
    stack < int > ancestors;
    int root = nodes[0];
    ancestors.push(root);
    for(int i = 1; i < len; i++){
        while(!is_ancestor(ancestors.top(), nodes[i]))
            ancestors.pop();
        int p = ancestors.top();
        aux[p].push_back(nodes[i]);
        ancestors.push(nodes[i]);
    }
    ans = 0; dfs(root);
    printf("%d\n", ans);
    for(int node : nodes) aux[node].clear();
}

int main(){
    cin >> n;
    for(int i = 1; i < n; i++){
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    dfs_prep(1, 1);
    cin >> q;
    while(q--){
        int k;
        cin >> k;
        for(int i = 1; i <= k; i++){
            int node;
            cin >> node;
            important[node] = true;
            nodes.push_back(node);
        }
        bool rekt = false;
        for(int node : nodes){
            if((parent[node] != node) && (important[parent[node]]))
                rekt = true;
        }
        solve(rekt);
        for(int node : nodes) important[node] = false;
        nodes.clear();
    }
}
```

5.2 Block Cut Tree

5.2.1 Block Cut Tree

```
// Codeforces - Tourists (Some Div 1E)
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9 + 333;
const int MAX = (1 << 18);
const int LN  = 18;

int n, m, q, timekeeper, cnt_bcc, sz, w[MAX];
int disc[MAX], low[MAX], cut[MAX], bcc[MAX], baap[MAX], is_cut[MAX];
int tree[MAX << 1], timer, chainPos[MAX], head[MAX], root;
int sub[MAX], depth[MAX], dp[LN][MAX], vis[MAX];
vector < pair < int, int > > adj[MAX];
vector < int > bc_tree[MAX], in_bcc[MAX];
pair < int, int > temp[MAX];
multiset < int > costs[MAX];

inline void make_tree(int p_id = -1, int p = 0, int x = 1) {
    vis[x] = 1;
    temp[sz++] = make_pair(x, p);
    low[x] = disc[x] = ++timekeeper;
    for(auto it : adj[x]) {
        int u = it.first;
        int e = it.second;
        if(e != p_id) {
            if(!vis[u]) {
                make_tree(e, x, u);
                low[x] = min(low[x], low[u]);
                if(low[u] >= disc[x]) {
                    cut[x] = 1;
                    cnt_bcc++;
                    while(temp[sz] != make_pair(u, x)) {
                        bcc[temp[sz - 1].first] = cnt_bcc;
                        bcc[temp[sz - 1].second] = cnt_bcc;
                        sz--;
                    }
                    in_bcc[x].push_back(cnt_bcc);
                    baap[cnt_bcc] = x;
                }
            }
            else
                low[x] = min(low[x], disc[u]);
        }
    }
}

inline void pre(int p = 0, int x = root) {
    sub[x] = 1;
    dp[0][x] = p;
    depth[x] = depth[p] + 1;
    for(int i = 1; i < LN; i++) dp[i][x] = dp[i - 1][dp[i - 1][x]];
    for(auto it : bc_tree[x]) {
        int u = it;
        if(u != p) {
            pre(x, u);
            sub[x] += sub[u];
        }
    }
}

inline void hld(int p = -1, int x = root, int h = root) {
    head[x] = h;
    chainPos[x] = ++timer;
    int rdcount = -1, rajat = -1; // pro-child
    for(auto it : bc_tree[x]) {
        int u = it;
        if(u != p and sub[u] > rdcount) {
            rdcount = u;
            rajat = u;
        }
    }
    if(rajat != -1) hld(x, rajat, h);
    for(auto it : bc_tree[x]) {
        int u = it;
        if(u != p and u != rajat)
            hld(x, u, u);
    }
}

inline void update(int x, int k) {
    tree[x += MAX] = k;
    while(x > 1) {
        x >>= 1;
        tree[x] = min(tree[x + x], tree[x + x + 1]);
    }
}

inline int query(int l, int r) {
    int res = INF;
    for(l += MAX, r += MAX; l <= r; l >>= 1, r >>= 1) {
        if(l & 1) res = min(res, tree[l++]);
        if(~r & 1) res = min(res, tree[r--]);
    }
    return res;
}

inline int get_lca(int x, int y) {
    if(depth[x] < depth[y]) swap(x, y);
    for(int i = LN - 1; i >= 0; i--)
        if(depth[x] - (1 << i) >= depth[y])
            x = dp[i][x];
    if(x == y) return x;
    for(int i = LN - 1; i >= 0; i--) {
        if(dp[i][x] != dp[i][y]) {
            x = dp[i][x];
            y = dp[i][y];
        }
    }
    return dp[0][x];
}

inline int qmin(int x, int up) {
    int res = INF;
    while(depth[x] >= depth[up]) {
        res = min(res, query(max(chainPos[up], chainPos[head[x]]), chainPos[x]));
        x = dp[0][head[x]];
    }
    return res;
}

int main () {

    scanf("%d %d %d", &n, &m, &q);

    for(int i = 1; i <= n; i++) {
        scanf("%d", w + i);
    }

    for(int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d %d", &x, &y);
        adj[x].push_back(make_pair(y, i));
        adj[y].push_back(make_pair(x, i));
    }
```

```
make_tree();

for(int i = n; i >= 1; i--) {
    if(cut[i]) {
        cut[i] = ++cnt_bcc;
        is_cut[cnt_bcc] = i;
    }
}

root = cnt_bcc;

for(int i = 1; i <= n; i++) {
    if(cut[i]) {
        bc_tree[bcc[i]].push_back(cut[i]);
        bc_tree[cut[i]].push_back(bcc[i]);
        for(auto it : in_bcc[i]) {
            int x = it;
            bc_tree[cut[i]].push_back(x);
            bc_tree[x].push_back(cut[i]);
        }
    }
}

for(int i = 1; i <= n; i++) {
    sort(bc_tree[i].begin(), bc_tree[i].end());
    bc_tree[i].resize(unique(bc_tree[i].begin(), bc_tree[i].end()) - bc_tree[i].begin());
}

pre();
hld();

for(int i = 1; i <= n; i++) {
    costs[bcc[i]].insert(w[i]);
}

for(int i = 1; i <= cnt_bcc; i++) {
    update(chainPos[i], costs[i].size() ? *costs[i].begin() : INF);
}

while(q--){
    char c;
    int x, y;
    scanf(" %c %d %d", &c, &x, &y);
    if(c == 'A') {
        if(x == y) {
            printf("%d\n", w[x]);
            continue;
        }
        x = cut[x] ? cut[x] : bcc[x];
        y = cut[y] ? cut[y] : bcc[y];
        int lca = get_lca(x, y);
        int res = min(qmin(x, lca), qmin(y, lca));
        if(is_cut[lca]) res = min(res, w[is_cut[lca]]);
        else if(baap[lca]) res = min(res, w[baap[lca]]);
        printf("%d\n", res);
    }
    else {
        costs[bcc[x]].erase(costs[bcc[x]].find(w[x]));
        costs[bcc[x]].insert(w[x] = y);
        update(chainPos[bcc[x]], *costs[bcc[x]].begin());
    }
    }
}
```

5.3 Bridge Tree

5.3.1 Bridge Tree

```
// Can you add one edge to a connected graph (with multiedges), to remove all bridges in it?
// This implementation handles multi-edges.
#include "bits/stdc++.h"
using namespace std;

const int N = 1e5 + 5;

int n, m, cur_time, component_id;
int a[N * 2], b[N * 2], is_bridge[N * 2];
int disc[N], low[N], component[N];
vector < int > adj[N], tree[N];

inline void find_bridges(int u, int p){
    disc[u] = low[u] = ++cur_time;
    for(int i = 0; i < (int) adj[u].size(); i++){
        int edge_id = adj[u][i];
        int v = a[edge_id] ^ b[edge_id] ^ u;
        if(edge_id == p) continue;
        if(!disc[v]){
            find_bridges(v, edge_id);
            low[u] = min(low[u], low[v]);
            if(low[v] > disc[u]) is_bridge[edge_id] = true;
        }
        else low[u] = min(low[u], disc[v]);
    }
}

inline void explore(int u){
    component[u] = component_id;
    for(int i = 0; i < (int) adj[u].size(); i++){
        int edge_id = adj[u][i];
        int v = a[edge_id] ^ b[edge_id] ^ u;
        if(component[v] || is_bridge[edge_id])  continue;
        explore(v);
    }
}

int main(){
    cin >> n >> m;
    for(int i = 1; i <= m; i++){
        cin >> a[i] >> b[i];
        adj[a[i]].push_back(i);
        adj[b[i]].push_back(i);
    }
    find_bridges(1, 0);
    for(int i = 1; i <= n; i++){
        if(!component[i]){
            ++component_id;
            explore(i);
        }
    }
    for(int i = 1; i <= m; i++){
        if(is_bridge[i]){
            int u = component[a[i]], v = component[b[i]];
            tree[u].push_back(v);
            tree[v].push_back(u);
        }
    }
    int cnt_leaves = 0;
    for(int i = 1; i <= component_id; i++){
        cnt_leaves += (((int) tree[i].size()) == 1);
    }
    if(cnt_leaves <= 2) cout << "YES\n";
    else cout << "NO\n";
}
```

5.4 Centroid Decomposition

5.4.1 Centroid Decomposition (Race)

```
// IOI 2011 - Race
#include "bits/stdc++.h"
using namespace std;

const int N = 1000000 + 50;
const int INF = 100000000;

int n, k, sub[N];
vector < pair < int, int > > adj[N];
vector < pair < int, int > > paths;
map < int, int > ans;
bool vis[N];

inline void dfs(int u, int p) {
    sub[u] = 1;
    for (int i = 0; i < (int) adj[u].size(); i++) {
        int v = adj[u][i].first;
        if ((v == p) or (vis[v])) {
            continue;
        }
        dfs(v, u);
        sub[u] += sub[v];
    }
}

inline int find_centroid(int u, int x, int p) {
    for (int i = 0; i < (int) adj[u].size(); i++) {
        int v = adj[u][i].first;
        if ((v == p) or (vis[v])) {
            continue;
        }
        if (sub[v] > x) {
            return find_centroid(v, x, u);
        }
    }
    return u;
}

inline void explore(int u, int p, int cost, int edges) {
    if (cost <= k) {
        paths.push_back(make_pair(cost, edges));
    }
    for (int i = 0; i < (int) adj[u].size(); i++) {
        int v = adj[u][i].first;
        int w = adj[u][i].second;
        if ((vis[v]) or (cost + w > k) or (v == p)) {
            continue;
        }
        explore(v, u, cost + w, edges + 1);
    }
}

inline int best_path(int u) {
    ans.clear();
    ans[0] = 0, ans[k] = INF;
    for (int i = 0; i < (int) adj[u].size(); i++) {
        int v = adj[u][i].first;
        int w = adj[u][i].second;
        if (vis[v]) {
            continue;
        }
        paths.clear();
        explore(v, u, w, 1);
        for (int j = 0; j < (int) paths.size(); j++) {
            int path_cost = paths[j].first;
            int edges_used = paths[j].second;
            if (ans.find(k - path_cost) != ans.end()) {
                ans[k] = min(ans[k], ans[k - path_cost] + edges_used);
            }
        }
        for (int j = 0; j < (int) paths.size(); j++) {
            int path_cost = paths[j].first;
            int edges_used = paths[j].second;
            if (ans.find(path_cost) == ans.end()) {
                ans[path_cost] = edges_used;
            } else {
                ans[path_cost] = min(ans[path_cost], edges_used);
            }
        }
    }
    return ans[k];
}

inline int decompose(int root, int p){
    int u = find_centroid(root, sub[root] >> 1, p);
    vis[u] = true;
    int res = best_path(u);
    for (int i = 0; i < (int) adj[u].size(); i++) {
        int v = adj[u][i].first;
        if (vis[v]) {
            continue;
        }
        dfs(v, u);
        res = min(res, decompose(v, u));
    }
    return res;
}

int main(){
    scanf("%d %d", &n, &k);
    for (int i = 1; i < n; i++) {
        int u, v, w; scanf("%d %d %d", &u, &v, &w);
        adj[u].push_back(make_pair(v, w));
        adj[v].push_back(make_pair(u, w));
    }
    dfs(0, -1);
    int res = decompose(0, -1);
    if (res > n) {
        res = -1;
    }
    printf("%d\n", res);
}
```

5.4.2 Centroid Subtrees

```
/*
   Computing centroid for every subtree of a tree.
 */

#include "bits/stdc++.h"
using namespace std;

const int MAX = 3e5 + 5;

int par[MAX], centroid[MAX];
long long len[MAX];
long long sub[MAX], tot[MAX];
long long dist_to_centroid[MAX], ans[MAX];
long long res = 0;
vector < int > adj[MAX];

/*
   Let s(v) denote the subtree rooted at (v)
   The tree is a directed tree with its root at 0.
   centroid[u] = centroid of s(u)
   ans[u] = sum(dist(centroid[u], v)) for all v in s(u)
   len[u] = weight of edge from u to parent[u]
   sub[u] = subtree size of s(u)
   tot[u] = sum(dist(u, v)) for all v in s(u)
   dist_to_centroid[u] = distance from u to centroid of s(u)
 */

struct TreeSums{
    int find_upper(int root, int u){
        int upper_tree = sub[root] - sub[u];
        return upper_tree;
    }
    void dfs_find(int u){
        centroid[u] = u;
        int heavy_child = -1;
        for(int v : adj[u]){
            dfs_find(v);
            if((heavy_child == -1) || (sub[v] > sub[heavy_child]))
                heavy_child = v;
        }
        if(heavy_child == -1) return; // Leaf node
        if(sub[heavy_child] <= sub[u] >> 1){ // u is the centroid of s(u)
            ans[u] = tot[u];
            res ^= ans[u];
            return;
        }
        /*
           The centroid is in s(heavy_child)
           We have recursively computed the centroid of s(heavy_child)
           The centroid for s(u) is now some ancestor of the centroid
```

```
             of s(heavy_child), or simply the centroid of s(heavy_child).
             Hence, we try to raise or lift the centroid of s(heavy_child)
             until the size of the "upper_tree" becomes <= s(u) / 2
          */
        centroid[u] = centroid[heavy_child];
        dist_to_centroid[u] = dist_to_centroid[heavy_child] + len[heavy_child];
        ans[u] = ans[heavy_child] + dist_to_centroid[u] * (sub[u] - sub[heavy_child])
            + tot[u] - (tot[heavy_child] + len[heavy_child] * sub[heavy_child]);
        while(true){
            int upper_tree = find_upper(u, centroid[u]);
            if(upper_tree <= sub[u] >> 1) break;
            ans[u] -= (2 * upper_tree - sub[u]) * len[centroid[u]];
            dist_to_centroid[u] -= len[centroid[u]];
            centroid[u] = par[centroid[u]];
        }
        res ^= ans[u];
    }
    void dfs_prep(int u){
        sub[u] = 1;
        for(int v : adj[u]){
            dfs_prep(v);
            sub[u] += sub[v];
            tot[u] += (tot[v] + len[v] * sub[v]);
        }
    }
    long long findSums(int N, int seed, int C, int D){
        for(int i = 0; i <= N; i++){
            adj[i].clear();
            sub[i] = tot[i] = ans[i] = dist_to_centroid[i] = 0;
            res = 0;
        }
        long long cur = seed;
        for(int i = 0; i <= N - 2; i++){
            cur = (C * 1LL * cur + D) % 1000000000;
            par[i + 1] = cur % (i + 1);
            adj[par[i + 1]].push_back(i + 1);
            cur = (C * 1LL * cur + D) % 1000000000;
            len[i + 1] = cur % 1000000;
        }
        dfs_prep(0);
        dfs_find(0);
        return res;
    }
};

int main(){
    TreeSums obj;
    cout << obj.findSums(6, 8, 3, 13) << '\n';
}
```

5.4.3 Tree Queries Centroid

```
/*
   Given a Tree T and Q queries, each of the form (v, l) :
   Each query returns number of vertices u such that distance(v, u) <= l

   Build the centroid tree, and handle each query in O(log n).
 */
```

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

const int MAX = 100005;
const int  LN = 20;

int n, q;
vector < pair < int, long long > > adj[MAX];
int done[MAX], parent[MAX], depth[MAX], sub[MAX];
long long dist[LN][MAX];
vector < long long > val[MAX], valp[MAX];

void dfs(int u, int p){
    sub[u] = 1;
    for(auto v : adj[u]){
        if(v.first == p || done[v.first]) continue;
        dfs(v.first, u);
        sub[u] += sub[v.first];
    }
}

int find(int u, int p, int tar){
    for(auto v : adj[u]){
        if(v.first == p || done[v.first]) continue;
        if(sub[v.first] > tar) return find(v.first, u, tar);
    }
    return u;
}

void explore(int u, int p, long long d, int cur){
    val[cur].push_back(d);
    dist[depth[cur]][u] = d;
    for(auto v : adj[u]){
        if(done[v.first] || v.first == p) continue;
        explore(v.first, u, d + v.second, cur);
    }
}

void decompose(int u, int p){
    dfs(u, p);
    int centroid = find(u, p, sub[u] / 2);
    parent[centroid] = p;
    done[centroid]   = true;
    depth[centroid]  = (p == 0) ? (0) : (depth[p] + 1);
    explore(centroid, p, 0, centroid);
    sort(val[centroid].begin(), val[centroid].end());
    for(auto v : adj[centroid]){
        if(done[v.first]) continue;
        decompose(v.first, centroid);
    }
}

void preprocess(){
    for(int i = 1; i <= n; i++){
        int cur = i;
        while(parent[cur] != 0){
            valp[cur].push_back(dist[depth[parent[cur]]][i]);
            cur = parent[cur];
        }
    }
    for(int i = 1; i <= n; i++) sort(valp[i].begin(), valp[i].end());
}
```

```
int query(int v, long long l){
    int ans = upper_bound(val[v].begin(), val[v].end(), l) - val[v].begin();
    int cur = v;
    while(parent[cur] != 0){
        long long d = dist[depth[parent[cur]]][v];
        int tot = upper_bound(val[parent[cur]].begin(), val[parent[cur]].end(), l - d) - val[parent[cur]].begin();
        int ext = upper_bound(valp[cur].begin(), valp[cur].end(), l - d) - valp[cur].begin();
        ans += tot - ext;
        cur = parent[cur];
    }
    return ans;
}

int main(){

    scanf("%d %d\n", &n, &q);
    for(int i = 1; i < n; i++){
        int u, v;
        long long l;
        scanf("%d %d %lld\n", &u, &v, &l);
        adj[u].push_back(make_pair(v, l));
        adj[v].push_back(make_pair(u, l));
    }

    decompose(1, 0);
    preprocess();
    while(q--){
        int v;
        long long l;
        scanf("%d %lld\n", &v, &l);
        printf("%d\n", query(v, l));
    }
}
```

5.4.4 YATP

```
// NAIPC 2016: YATP
// Centroid Decomposition + Convex hull trick

#include "bits/stdc++.h"
using namespace std;

struct convexHullTrick{
    struct Line{
        long long a , b;
        double xleft;
        bool type;
        Line(long long _a , long long _b){
            a = _a;
            b = _b;
            type = 0;
        }
        bool operator < (const Line &other) const{
            if(other.type){
                return xleft < other.xleft;
            }
            return a > other.a;
        }
    };
```

```
inline double intersect(Line x , Line y){
    return 1.0 * (y.b - x.b) / (x.a - y.a);
}
multiset < Line > hull;
convexHullTrick(){
    hull.clear();
}
typedef set < Line > :: iterator iter;
inline bool hasLeft(iter node){
    return node != hull.begin();
}
inline bool hasRight(iter node){
    return node != prev(hull.end());
}
inline void updateBorder(iter node){
    if(hasRight(node)){
        Line temp = *next(node);
        hull.erase(hull.find(temp));
        temp.xleft = intersect(*node, temp);
        hull.insert(temp);
    }
    if(hasLeft(node)){
        Line temp = *node;
        temp.xleft = intersect(*prev(node), temp);
        hull.erase(node);
        hull.insert(temp);
    }
    else{
        Line temp = *node;
        hull.erase(node);
        temp.xleft = -1e18;
        hull.insert(temp);
    }
}
inline bool isBad(Line left , Line middle , Line right){
    return intersect(left, middle) > intersect(middle, right);
}
inline bool isBad(iter node){
    if(hasLeft(node) && hasRight(node)){
        return isBad(*prev(node), *node, *next(node));
    }
    return 0;
}
// add Line with equation y = (a * x + b)
inline void addLine(long long a, long long b){
    Line temp = Line(a, b);
    auto it = hull.lower_bound(temp);
    if(it != hull.end() && it -> a == a){
        if(it -> b > b){
            hull.erase(it);
        }
        else{
            return;
        }
    }
    hull.insert(temp);
    it = hull.find(temp);
    if(isBad(it)){
        hull.erase(it);
        return;
    }
    while(hasLeft(it) && isBad(prev(it))){
            hull.erase(prev(it));
        }
        while(hasRight(it) && isBad(next(it))){
            hull.erase(next(it));
        }
        updateBorder(it);
    }
    // get minimum value of (m * x + c) for given x
    inline long long getBest(long long x){
        if(hull.empty()){
            return 5e18;
        }
        Line getBest(0, 0);
        getBest.xleft = x;
        getBest.type = 1;
        auto it = hull.lower_bound(getBest);
        it = prev(it);
        return it -> a * x + it -> b;
    }
};

const int N = 200000 + 50;
int n, a, b, c, sz;
int penalty[N], subtreeSize[N];
bool done[N];
long long bestValue[N];
vector < pair < int, int > > adj[N];
convexHullTrick dp;

inline void dfsPre(int node, int parent) {
    subtreeSize[node] = 1;
    for (auto next : adj[node]) {
        if ((next.first != parent) && (!done[next.first])) {
            dfsPre(next.first, node);
            subtreeSize[node] += subtreeSize[next.first];
        }
    }
}

inline int findCentroid(int node, int parent) {
    for (auto next : adj[node]) {
        if ((next.first != parent) && (!done[next.first])) {
            if (subtreeSize[next.first] > sz) {
                return findCentroid(next.first, node);
            }
        }
    }
    return node;
}

inline void exploreChild(int node, int parent, long long dist) {
    bestValue[node] = min(bestValue[node], dp.getBest(penalty[node]) + dist);
    for (auto next : adj[node]) {
        if ((next.first != parent) && (!done[next.first])) {
            exploreChild(next.first, node, dist + next.second);
        }
    }
}

inline void addChild(int node, int parent, long long dist) {
    dp.addLine(penalty[node], dist);
    for (auto next : adj[node]) {
        if ((next.first != parent) && (!done[next.first])) {
            addChild(next.first, node, dist + next.second);
        }
    }
}

inline void solve(int node) {
    dp.hull.clear();
    dp.addLine(penalty[node], 0);
    for (int i = 0; i < (int) adj[node].size(); i++) {
        pair < int, int > next = adj[node][i];
        if (!done[next.first]) {
            exploreChild(next.first, node, next.second);
            addChild(next.first, node, next.second);
        }
    }
    dp.hull.clear();
    reverse(adj[node].begin(), adj[node].end());
    dp.addLine(penalty[node], 0);
    for(int i = 0; i < (int) adj[node].size(); i++){
        pair < int, int > next = adj[node][i];
        if(!done[next.first]){
            exploreChild(next.first, node, next.second);
            addChild(next.first, node, next.second);
        }
    }
    bestValue[node] = min(bestValue[node], dp.getBest(penalty[node]));
}

inline void decompose(int node) {
    dfsPre(node , 0);
    sz = (subtreeSize[node] >> 1);
    int centroid = findCentroid(node, 0);
    solve(centroid);
    done[centroid] = 1;
    for (auto next : adj[centroid]) {
        if (!done[next.first]) {
            decompose(next.first);
        }
    }
}

int main() {
    freopen("inp.in", "r", stdin);
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", penalty + i);
        bestValue[i] = penalty[i] * 1LL * penalty[i];
    }
    for (int i = 1; i < n; i++) {
        scanf("%d %d %d", &a, &b, &c);
        adj[a].emplace_back(make_pair(b, c));
        adj[b].emplace_back(make_pair(a, c));
    }
    decompose(1);
    long long ans = 0;
    for (int i = 1; i <= n; i++) {
        ans += bestValue[i];
    }
    printf("%lld\n", ans);
}
```

5.5 DAG

5.5.1 Dominators (DAG)

```
/*
   Given Directed Graph G = (V, E), and a source s.
   f(s, x) : Number of nodes y such that all shortest paths from s to x pass through y.
Problem : Find the node x (x != s) in G that has the maximum f(s, x).

Solution uses concept of Dominators but is simpler since the shortest path DAG we consider
is acyclic. For general graphs, we will need a Dominator Tree.

Problem Source : CodeCraft 2017 'F'.
 */
#include "bits/stdc++.h"
using namespace std;

const int N = 200000 + 10;
const int M = 300000 + 10;
const long long INF = (1LL << 60LL);

int n, m, s;
int u[M], v[M], w[M];
vector < pair < int, int > > adj[N];
vector < int > parents[N];
int topological_order[N];
long long dist[N];
int hdom[N], ans[N];

inline void dijkstra() {
    set < pair < long long, int > > dijkstra;
    for (int i = 1; i <= n; i++) {
        dist[i] = INF;
    }
    dijkstra.insert({0, s});
    int ticks = 0;
    while (!dijkstra.empty()) {
        long long curDistance = dijkstra.begin() -> first;
        int curNode = dijkstra.begin() -> second;
        dijkstra.erase(dijkstra.begin());
        if (dist[curNode] == INF) {
            dist[curNode] = curDistance;
            topological_order[++ticks] = curNode;
            for (auto edge : adj[curNode]) {
                dijkstra.insert({curDistance + edge.second, edge.first});
            }
        }
    }
    for (int i = 1; i <= m; i++) {
        if (dist[u[i]] + w[i] == dist[v[i]]) {
            parents[v[i]].push_back(u[i]);
        }
        if (dist[v[i]] + w[i] == dist[u[i]]) {
            parents[u[i]].push_back(v[i]);
        }
    }
}

inline void solve() {
    int res = 0;
    for (int i = 2; i <= n; i++) {
        int cur = topological_order[i];
        if (cur == 0) {
            break;
        }
        bool ok = true;
        int prev = -1;
        for (int p : parents[cur]) {
            if (p == s) {
                ok = false;
            } else if ((prev != -1) && (hdom[p] != prev)) {
                ok = false;
            }
            prev = hdom[p];
        }
        if (ok) {
            hdom[cur] = prev;
        } else {
            hdom[cur] = cur;
        }
        ans[hdom[cur]]++;
        res = max(res, ans[hdom[cur]]);
    }
    cout << res << "\n";
}

int main() {
    cin >> n >> m >> s;
    for (int i = 1; i <= m; i++) {
        cin >> u[i] >> v[i] >> w[i];
        adj[u[i]].push_back({v[i], w[i]});
        adj[v[i]].push_back({u[i], w[i]});
    }
    dijkstra();
    solve();
}
```

5.5.2 Semidynamic Longest Path

```
/*
   ans[u] = Longest path in a DAG after removing vertex (u)
   POI Rally
 */

#include "bits/stdc++.h"
using namespace std;

const int N = 5e5 + 5;
const int INF = 1e8 + 8;

int n, m;
int in_degree[N], order[N], pos[N];
int in[N], out[N], ans[N];
int tree[N * 4];
vector < int > adj[N];

inline void topological_sort(){
    int cur_time = 0;
    queue < int > q;
    for(int i = 1; i <= n; i++){
        if(!in_degree[i]) q.push(i);
    }

    while(!q.empty()){
        int u = q.front();
        q.pop();
        order[++cur_time] = u;
        pos[u] = cur_time;
        for(int j = 0; j < (int) adj[u].size(); j++){
            int v = adj[u][j];
            in_degree[v]--;
            if(!in_degree[v]) q.push(v);
        }
    }

    for(int i = 1; i <= n; i++){
        int u = order[i];
        for(int j = 0; j < (int) adj[u].size(); j++){
            int v = adj[u][j];
            in[v] = max(in[v], in[u] + 1);
        }
    }

    for(int i = n; i >= 1; i--){
        int u = order[i];
        for(int j = 0; j < (int) adj[u].size(); j++){
            int v = adj[u][j];
            out[u] = max(out[u], out[v] + 1);
        }
    }
}

inline void update(int i, int l, int r, int qs, int qe, int val){
    if(l > qe || r < qs) return;
    if(l >= qs && r <= qe){
        tree[i] = max(tree[i], val);
        return;
    }
    int mid = (l + r) >> 1;
    update(i * 2, l, mid, qs, qe, val);
    update(i * 2 + 1, mid + 1, r, qs, qe, val);
}

inline int query(int i, int l, int r, int pos){
    if(l == r) return tree[i];
    int mid = (l + r) >> 1;
    if(mid >= pos) return max(tree[i], query(i * 2, l, mid, pos));
    else return max(tree[i], query(i * 2 + 1, mid + 1, r, pos));
}

int main(){
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= m; i++){
        int u, v;
        scanf("%d %d", &u, &v);
        adj[u].push_back(v);
        in_degree[v]++;
    }

    topological_sort();

    for(int u = 1; u <= n; u++){
        for(int j = 0; j < (int) adj[u].size(); j++){
            int v = adj[u][j];
            int l = pos[u] + 1, r = pos[v] - 1;
            if(l <= r) update(1, 1, n, l, r, in[u] + 1 + out[v]);
        }
    }

    for(int i = 1; i <= n; i++){
        ans[i] = query(1, 1, n, pos[i]);
    }

    int prefix_max = 0;
    for(int i = 2; i <= n; i++){
        int u = order[i];
        prefix_max = max(prefix_max, in[order[i - 1]]);
        ans[u] = max(ans[u], prefix_max);
    }

    int suffix_max = 0;
    for(int i = n - 1; i >= 1; i--){
        int u = order[i];
        suffix_max = max(suffix_max, out[order[i + 1]]);
        ans[u] = max(ans[u], suffix_max);
    }

    int min_val = INF, min_idx = -1;
    for(int i = 1; i <= n; i++){
        if(ans[i] < min_val){
            min_val = ans[i];
            min_idx = i;
        }
    }

    printf("%d %d\n", min_idx, min_val);
}
```

5.6 Directed MST

5.6.1 Edmonds

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

// Directed minimum spanning tree
//
// Given a directed weighted graph and root node, computes the minimum spanning
// directed tree (arborescence) on it.
//
// Complexity: O(N * M)


struct Edge { int x, y, w; };

int dmst(int N, vector<Edge> E, int root) {
    const int oo = 1e9;

    vector<int> cost(N), back(N), label(N), bio(N);
    int ret = 0;

    for (;;) {
        for(int i = 0; i < N; i++) cost[i] = oo;
        for (auto e : E) {
            if (e.x == e.y) continue;
            if (e.w < cost[e.y]) cost[e.y] = e.w, back[e.y] = e.x;
        }
        cost[root] = 0;
        for(int i = 0; i < N; i++) if (cost[i] == oo) return -1;
        for(int i = 0; i < N; i++) ret += cost[i];

        int K = 0;
        for(int i = 0; i < N; i++) label[i] = -1;
        for(int i = 0; i < N; i++) bio[i] = -1;

        for(int i = 0; i < N; i++) {
            int x = i;
            for (; x != root && bio[x] == -1; x = back[x]) bio[x] = i;

            if (x != root && bio[x] == i) {
                for (; label[x] == -1; x = back[x]) label[x] = K;
                ++K;
            }
        }
        if (K == 0) break;

        for(int i = 0; i < N; i++) if (label[i] == -1) label[i] = K++;

        for (auto &e : E) {
            int xx = label[e.x];
            int yy = label[e.y];
            if (xx != yy) e.w -= cost[e.y];
            e.x = xx;
            e.y = yy;
        }

        root = label[root];
        N = K;
    }

    return ret;
}

int main() {
    int t; cin >> t;
    for (int qq = 1; qq <= t; qq++) {
        cout << "Case " << qq << ": ";
        int n, m, k; cin >> n >> m >> k;
        vector < Edge > e(m);
        for (int i = 0; i < m; i++) {
            Edge cur;
            cin >> cur.x >> cur.y >> cur.w;
            e[i] = cur;
        }
        int res = dmst(n, e, k);
        if (res == -1) {
            cout << "impossible" << endl;
        } else {
            cout << res << endl;
        }
    }
}
```

5.7 Euler Path

5.7.1 Euler Path

```
/**
  The Algorithm below is known as Heirholzer's Algorithm. It runs in O(V + E).

  The algorithm finds an Euler Tour or an Euler Circuit depending on whether the
  graph has 0 or 2 nodes with odd degree. If the graph is not Eulerian, it still
  returns some list, so make sure to first check that the graph is Eulerian.

  A connected graph has an Euler Path (A path that uses all edges but starts and ends at
  different nodes) if it has exactly 2 nodes with odd degree. While calling find_path()
  make sure you ensure that you call it with an ODD degree node since the path must
  start/end there.

  A connected graph has an Euler Circuit (A circuit that uses all edges and starts and ends
  at the same node) if all nodes have even degree. You can call find_path() with any
  node here.

  This algorithm handles self loops and multiple edges as well, but only works on undirected
  graphs.
 **/

struct Edge;
typedef list<Edge>::iterator iter;

struct Edge
{
    int next_vertex;
    iter reverse_edge;

    Edge(int next_vertex)
        :next_vertex(next_vertex)
    { }
};

const int max_vertices = ;
int num_vertices;
list<Edge> adj[max_vertices];       // adjacency list

vector<int> path;

void find_path(int v)
{
    while(adj[v].size() > 0)
    {
        int vn = adj[v].front().next_vertex;
        adj[vn].erase(adj[v].front().reverse_edge);
        adj[v].pop_front();
        find_path(vn);
    }
    path.push_back(v);
}

void add_edge(int a, int b)
{
    adj[a].push_front(Edge(b));
    iter ita = adj[a].begin();
    adj[b].push_front(Edge(a));
    iter itb = adj[b].begin();
    ita->reverse_edge = itb;
    itb->reverse_edge = ita;
}

// ************ End of Undirected Euler Path *************

/**
  A directed graph has an Eulerian cycle if and only if every node has an
  in-degree equal to its out-degree, and all of its nodes with nonzero degree
  belong to a single strongly connected component.

  A directed graph has an Eulerian path if and only if it is strongly connected and
  each vertex except 2 have the same in-degree as out-degree, and one of those 2
  vertices has out-degree with one greater than in-degree (this is the start vertex),
  and the other vertex has in-degree with one greater than out-degree (this is the end vertex).

  The code below works for both cases. For Euler Path, make sure that you call the function
  with the appropriate start vertex.
 **/

std::vector<int> find_path_directed(std::vector<int> adj[], int u) {
    std::vector<int> stack, curr_edge(MAXN), res;
    stack.push_back(u);
    while (!stack.empty()) {
        u = stack.back();
        stack.pop_back();
        while (curr_edge[u] < (int)adj[u].size()) {
            stack.push_back(u);
            u = adj[u][curr_edge[u]++];
        }
        res.push_back(u);
    }
    std::reverse(res.begin(), res.end());
    return res;
}

// ************ End of Directed Euler Path *************
```

5.7.2 Make Graph Eulerian

```
// CF - Story of Princess (ICPC 15/16 Samara)

#include "bits/stdc++.h"
using namespace std;

struct Edge;
typedef list<Edge>::iterator iter;

struct Edge
{
    int next_vertex;
    iter reverse_edge;

    Edge(int next_vertex)
        :next_vertex(next_vertex)
    { }
};

const int max_vertices = 200000;
int num_vertices;
```

```
list<Edge> adj[max_vertices];       // adjacency list

vector<int> path;

void find_path(int v)
{
    while(adj[v].size() > 0)
    {
        int vn = adj[v].front().next_vertex;
        adj[vn].erase(adj[v].front().reverse_edge);
        adj[v].pop_front();
        find_path(vn);
    }
    path.push_back(v);
}

void add_edge(int a, int b)
{
    adj[a].push_front(Edge(b));
    iter ita = adj[a].begin();
    adj[b].push_front(Edge(a));
    iter itb = adj[b].begin();
    ita->reverse_edge = itb;
    itb->reverse_edge = ita;
}

// ************ End of Undirected Euler Path *************

/**
  Essentially, we want an Eulerian Graph since those are the nicest for this problem.
  However, the graph we get might not be Eulerian. It can be shown that we can't get
  an optimal solution if the graph is not Eulerian.

  We can reduce the problem to : "Add fewest edges to make the graph Eulerian". This
  can be solved greedily. For each component, compute # of odd degree vertices, this
  number has to be even. Match them up until you have 2 vertices of odd degree. Now
  your graph is Eulerian, use Heirholzer to compute an Euler Path.
 **/

int n, m;
vector < int > odd;
int vis[max_vertices];

inline void dfs(int node) {
    vis[node] = true;
    int degree = 0;
    for (Edge e: adj[node]) {
        int next = e.next_vertex;
        degree += 1;
        if (!vis[next]) {
            dfs(next);
        }
    }
    if (degree & 1) {
        odd.push_back(node);
    }
}

inline void solve(int root) {
    odd.clear();
    dfs(root);
    int path_start = root;
    while ((int) odd.size() > 2) {
        int u = odd.back(); odd.pop_back();
        int v = odd.back(); odd.pop_back();
        add_edge(u, v);
    }
    if ((int) odd.size() == 2) {
        root = odd[0];
    }
    if ((int) adj[root].size()) {
        find_path(root);
    }
}

int main() {
    ios :: sync_with_stdio(false);
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v; cin >> u >> v;
        add_edge(u, v);
    }
    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            solve(i);
        }
    }
    cout << (int) path.size() << endl;
    for (int v: path) {
        cout << v << ' ';
    }
    cout << endl;
}
```

5.8 Gomory Hu Tree

5.8.1 Gomory Hu Tree

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

/*
   Template for Gomory Hu Tree Construction.
NMAX : Number of Nodes in the Graph
MMAX : Number of Edges in the Graph
INF : INFINITY (Should be greater than maximum flow possible)

We use Gusfield's Algorithm to construct the Gomory Hu Tree.
This uses n - 1 Max Flow computations. We use Edmonds' Karp Algorithm
for the Max Flow. Substituting it with Dinics' would make code faster.

Make sure the variable 'n' holds number of nodes in the graph (which MUST be indexed 1 to n).
Similarly, 'm' should hold number of edges in the graph. The variables 's'
and 't' should not used elsewhere in your code. The function accepts the
adjacency matrix of the graph which must be stored in the global 'adj'.
It can handle both undirected and directed graphs. To handle multiedges,
increment edge weights appropriately (suppose there are c edges between u to v with cost y,
then adj[u][v] = c * y).

The constructed Gomory Hu tree is stored in 'tree'. min_cut[i][j] holds the value of
the min_cut between nodes (i) and (j).
```

```
This implementation assumes everything, included all max flow values, fit in 32-bit ints.
To deal with long longs, the easiest fix would be #define int long long and replace INF
with 1e18.
 */

const int NMAX = 205;
const int MMAX = 1005;
const int INF = 1e9;

int n, m, s, t;
int cap[NMAX][NMAX], flow[NMAX][NMAX], adj[NMAX][NMAX];
int min_cut[NMAX][NMAX];
int par[NMAX], par_in_tree[NMAX];
bool vis[NMAX];
vector < pair < int, int > > tree[NMAX];
vector < int > graph[NMAX];
queue < int > bfs;

// Finds several augmenting paths at once
inline bool augmenting_paths() {
    memset(par, 0, sizeof par);
    memset(vis, 0, sizeof vis);
    bfs.push(s);
    vis[s] = true;
    par[s] = -1;
    int node;
    while (!bfs.empty()) {
        node = bfs.front();
        bfs.pop();
        for (auto it: graph[node]) {
            if (cap[node][it] - flow[node][it] && !vis[it]) {
                par[it] = node;
                vis[it] = true;
                bfs.push(it);
            }
        }
    }
    return vis[t];
}

inline int edmonds_karp() {
    memset(flow, 0, sizeof flow);
    int ans = 0;
    while (augmenting_paths()) {
        for (int i = 1; i <= n; ++i) {
            if ((cap[i][t] - flow[i][t]) && (vis[i])) {
                int node = i;
                int cur_flow = cap[i][t] - flow[i][t];
                while (node != s) {
                    cur_flow = min(cur_flow, cap[par[node]][node] - flow[par[node]][node]);
                    node = par[node];
                }
                flow[i][t] += cur_flow;
                flow[t][i] -= cur_flow;
                node = i;
                ans += cur_flow;
                while (node != s) {
                    flow[par[node]][node] += cur_flow;
                    flow[node][par[node]] -= cur_flow;
                    node = par[node];
                }
            }
        }
    }
    return ans;
}

inline void constructGomoryHuTree(int cur[NMAX][NMAX]) {
    // Cleanup Graphs and Trees, Reset Necessary Stuff
    for (int i = 1; i <= n; i++) {
        graph[i].clear();
        tree[i].clear();
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            min_cut[i][j] = INF;
        }
    }
    for (int i = 2; i <= n; i++) {
        par_in_tree[i] = 1;
    }
    // Build Graph
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            cap[i][j] = cur[i][j];
            if (cap[i][j]) {
                graph[i].push_back(j);
            }
        }
    }
    // Gusfield's Algorithm to Compute the Gomory Hu Tree
    for (int i = 2; i <= n; i++) {
        s = i, t = par_in_tree[i];
        min_cut[i][par_in_tree[i]] = min_cut[par_in_tree[i]][i] = edmonds_karp();
        int edge_weight = min_cut[i][par_in_tree[i]];
        tree[t].push_back(make_pair(s, edge_weight));
        tree[s].push_back(make_pair(t, edge_weight));
        for (int j = i + 1; j <= n; j++) {
            if ((vis[j]) && (par_in_tree[j] == t)) {
                par_in_tree[j] = i;
            }
        }
        for (int j = 1; j < i; j++) {
            min_cut[i][j] = min_cut[j][i] = min(min_cut[par_in_tree[i]][j], min_cut[i][par_in_tree[i]]);
        }
    }
}

// End of Template

int forbidden[NMAX];

inline void find_min_edge(int u, int p, int &min_val, int &min_u, int &min_v) {
    for (auto edge : tree[u]) {
        int v = edge.first;
        int w = edge.second;
        if ((forbidden[v]) || (v == p)) {
            continue;
        }
        find_min_edge(v, u, min_val, min_u, min_v);
        if (min_val > w) {
            min_val = w;
            min_u = u;
            min_v = v;
        }
    }
}

inline void solve(int node) {
    int min_u = INF, min_v = INF, min_val = INF;
    find_min_edge(node, 0, min_val, min_u, min_v);
    if (min_val == INF) {
        cout << node << ' ';
        return;
    }
    // Solve for one side
    forbidden[min_u] = true;
    solve(min_v);
    forbidden[min_u] = false;
    // Merge with other side
    forbidden[min_v] = true;
    solve(min_u);
    forbidden[min_v] = false;
}

int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int u, v, w; cin >> u >> v >> w;
        adj[u][v] = w;
        adj[v][u] = w;
    }
    constructGomoryHuTree(adj);
    int cost = 0;
    for (int u = 1; u <= n; u++) {
        for (auto edge : tree[u]) {
            cost += edge.second;
        }
    }
    cost >>= 1;
    cout << cost << "\n";
    fill(forbidden + 1, forbidden + 1 + n, false);
    solve(1);
}
```

5.9 Heavy Light Decomposition

5.9.1 QTREE HLD

```
// You are given a tree (an acyclic undirected connected graph) with N nodes, and edges numbered 1, 2, 3...N-1.
// CHANGE i ti : change the cost of the i-th edge to ti
// QUERY a b : ask for the maximum edge cost on the path from node a to node b

#include <bits/stdc++.h>
#define rf freopen("inp.in", "r", stdin)
using namespace std;

const int MAX = 10005;

int t, n, x, y, a[MAX], b[MAX], c[MAX];
int depth[MAX], heavy[MAX], root[MAX], parent[MAX], sub[MAX];
int edgeToNode[MAX], nodeToEdge[MAX], pos[MAX], tree[MAX << 2];
vector < pair < int, int > > adj[MAX];
char str[MAX];
```

```
inline void dfs(int u, int p){
    sub[u] = 1;
    int mx = 0;
    for(int i = 0; i < adj[u].size(); i++){
        int v = adj[u][i].first;
        if(v == p) continue;
        edgeToNode[adj[u][i].second] = v;
        nodeToEdge[v] = adj[u][i].second;
        parent[v] = u;
        depth[v]  = depth[u] + 1;
        dfs(v, u);
        sub[u] += sub[v];
        if(sub[v] > mx){
            mx = sub[v];
            heavy[u] = v;
        }
    }
}

inline void update(int node, int l, int r, int idx, int val){
    if(l == r){
        tree[node] = val;
        return;
    }
    int mid = l + r >> 1;
    if(mid >= idx) update(node + node, l, mid, idx, val);
    else update(node + node + 1, mid + 1, r, idx, val);
    tree[node] = max(tree[node + node], tree[node + node + 1]);
}

inline int query(int node, int l , int r, int qs, int qe){
    if(l > qe or r < qs)    return 0;
    if(l >= qs and r <= qe) return tree[node];
    int mid = l + r >> 1;
    return max( query(node + node, l, mid, qs, qe), query(node + node + 1, mid + 1, r, qs, qe) );
}

inline void hld(){
    dfs(1, 0);
    for(int i = 1, curPos = 0; i <= n; i++){
        if(parent[i] == -1 || heavy[parent[i]] != i){
            for(int j = i; j != -1; j = heavy[j])
                root[j] = i, pos[j]  = ++curPos;
        }
    }
    for(int i = 2; i <= n; i++) update(1, 1, n, pos[i], c[nodeToEdge[i]]);
}

inline int query(int u, int v){
    int mx = 0;
    for(; root[u] != root[v]; v = parent[root[v]]){
        if(depth[root[u]] > depth[root[v]]) swap(u, v);
        mx = max(mx, query(1, 1, n, pos[root[v]], pos[v]));
    }
    if(depth[u] > depth[v]) swap(u, v);
    mx = max(mx, query(1, 1, n, pos[u] + 1, pos[v]));
    return mx;
}

inline void solve(){
    scanf("%d", &n);
    memset(tree, 0, sizeof tree);
    for(int i = 1; i <= n; i++){
        adj[i].clear();
        heavy[i] = parent[i] = edgeToNode[i] = nodeToEdge[i] = -1;
        depth[i] = sub[i] = 0;
    }
    for(int i = 1; i < n; i++){
        scanf("%d %d %d\n", &a[i], &b[i], &c[i]);
        adj[a[i]].push_back(make_pair(b[i], i));
        adj[b[i]].push_back(make_pair(a[i], i));
    }
    hld();
    while(true){
        scanf("%s", str);
        if(str[0] == 'D') break;
        else if(str[0] == 'C'){
            scanf("%d %d\n", &x, &y);
            update(1, 1, n, pos[edgeToNode[x]], y);
        }
        else{
            scanf("%d %d\n", &x, &y);
            printf("%d\n", query(x, y));
        }
    }
}

int main(){
    rf;
    scanf("%d", &t);
    while(t--) solve();
}
```

5.9.2 Subtrees and Paths HLD

```
// Given a rooted tree of  nodes, where each node is uniquely numbered in between [1..N].
// The node 1 is the root of the tree. Each node has an integer value which is initially 0.
// You need to perform the following two kinds of queries on the tree:
// add t value: Add value to all nodes in subtree rooted at t
// max a b: Report maximum value on the path from a to b

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

// Adjust appropriately.
const int MAX = 100005;
vector < int > adj[MAX];
int n, q, cur = 0;
int parent[MAX], root[MAX], heavy[MAX], depth[MAX], sub[MAX];
int tin[MAX], tout[MAX], pos[MAX];
int tree[MAX << 2], lazy[MAX << 2];

void dfs(int u, int p){
    sub[u] = 1;
    int mx = 0;
    for(int v : adj[u]){
        if(v == p) continue;
        parent[v] = u;
        depth[v]  = depth[u] + 1;
        dfs(v, u);
        sub[u] += sub[v];
        if(sub[v] > mx){
            mx = sub[v];
            heavy[u] = v;
        }
    }
}

void decompose(int u, int p){
    if(parent[u] == -1 || heavy[parent[u]] != u) root[u] = u;
    else root[u] = root[parent[u]];
    tin[u] = pos[u] = ++cur;
    if(heavy[u] != -1) decompose(heavy[u], u);
    for(int v : adj[u]){
        if(v == p or v == heavy[u]) continue;
        decompose(v, u);
    }
    tout[u] = cur;
}

void hld(){
    for(int i = 1; i <= n; i++) parent[i] = heavy[i] = -1;
    dfs(1, 0);
    decompose(1, 0);
}

void push(int l, int r, int node){
    tree[node] += (lazy[node]);
    if(l != r){
        lazy[node + node] += lazy[node];
        lazy[node + node + 1] += lazy[node];
    }
    lazy[node] = 0;
}

void update(int node, int l, int r, int qs, int qe, int val){
    push(l, r, node);
    if(l > qe or r < qs) return;
    if(l >= qs and r <= qe){
        lazy[node] = val;
        push(l, r, node);
        return;
    }
    int mid = l + r >> 1;
    update(node + node, l, mid, qs, qe, val);
    update(node + node + 1, mid + 1, r, qs, qe, val);
    tree[node] = max(tree[node + node], tree[node + node + 1]);
}

int query(int node, int l, int r, int qs, int qe){
    push(l, r, node);
    if(l > qe or r < qs) return -1e9;
    if(l >= qs and r <= qe) return tree[node];
    int mid = l + r >> 1;
    return max(query(node + node, l, mid, qs, qe), query(node + node + 1, mid + 1, r, qs, qe));
}

int query(int u, int v){
    int res = -1e9;
    for(; root[u] != root[v]; v = parent[root[v]]){
        if(depth[root[u]] > depth[root[v]]) swap(u, v);
        res = max(res, query(1, 1, n, pos[root[v]], pos[v]));
    }
    if(depth[u] > depth[v]) swap(u, v);
    res = max(res, query(1, 1, n, pos[u], pos[v]));
    return res;
}

int main(){
    scanf("%d\n", &n);
    for(int i = 1; i < n; i++){
        int u, v;
        scanf("%d %d\n", &u, &v);
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    scanf("%d\n", &q);
    hld();
    char buf[5];
    int x, y;
    while(q--){
        scanf("%s %d %d\n", buf, &x, &y);
        if(buf[0] == 'a'){
            update(1, 1, n, tin[x], tout[x], y);
        }
        else{
            printf("%d\n", query(x, y));
        }
    }
}
```

5.10 Reachability Tree

5.10.1 Reachability Tree

```
// Undirected Graph: Each node has a color, each edge has a weight.
// Q Queries: (u, v, k): Compute a (not necessarily simple) path from
// u to v, that visits at least k different colors, and minimizes the
// weight of the maximum edge on the path. N, Q <= 100000

// Solution: Construct the "reachability tree" of the given tree.
// We define the reachability tree as a rooted binary tree with 2N1
// nodes such that each leaf represents a node in the original tree,
// and each internal node represents a set of nodes in the original tree
// that are reachable from each other using edges up to a certain length.
// In other words, each internal node is associated with some value, say k
// such that the leaves under it represent a maximal set of nodes in the
// original tree that are reachable from each other using edges of length <= k
// This tree can be constructed in O(N log N)


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

const int N = 3e5 + 5;
const int LN = 21;

struct edge{
    int u, v, w;
    friend bool operator < (edge x, edge y){
        return (x.w < y.w);
    }
}edges[N];
```

```
int n, m, q, root;
vector < int > adj[N];
int t[N], val[N];
int tin[N], distinct[N], deg[N], parent[N];
int timer, head[N];
vector < int > values;
vector < int > nodes[N];
int depth[N], dp[N][LN];

inline int find(int x){
    if(parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}

inline int lca(int x, int y){
    if(depth[x] < depth[y]) swap(x, y);
    for(int i = LN - 1; i >= 0; i--){
        if(depth[x] - (1 << i) >= depth[y])
            x = dp[x][i];
    }
    if(x == y) return x;
    for(int i = LN - 1; i >= 0; i--){
        if(dp[x][i] != dp[y][i]){
            x = dp[x][i];
            y = dp[y][i];
        }
    }
    return dp[x][0];
}

inline void dfs_init(int u, int p, int rt){
    dp[u][0] = p;
    tin[u] = ++timer;
    head[u] = rt;
    for(int i = 1; i < LN; i++) dp[u][i] = dp[dp[u][i - 1]][i - 1];
    for(int v : adj[u]){
        if(v != p){
            depth[v] = depth[u] + 1;
            dfs_init(v, u, rt);
        }
    }
}

inline void dfs_find(int u, int p){
    for(int v : adj[u]){
        if(v != p){
            dfs_find(v, u);
            distinct[u] += distinct[v];
        }
    }
}

inline bool cmp(int x, int y){
    return (tin[x] < tin[y]);
}

inline void build(){
    sort(edges + 1, edges + 1 + m);
    for(int i = 1; i <= 3 * n; i++) parent[i] = i;
    root = n;
    for(int i = 1; i <= m; i++){
        int u = find(edges[i].u), v = find(edges[i].v), w = edges[i].w;
        if(u == v) continue;
        ++root;
        parent[u] = parent[v] = root;
        val[root] = w;
        adj[root].push_back(u);
        adj[root].push_back(v);
        deg[u]++, deg[v]++;
    }
    for(int i = root; i >= 1; i--){
        if(!deg[i]) dfs_init(i, i, i);
    }
    for(int i = 1; i <= n; i++){
        if(nodes[i].size()){
            sort(nodes[i].begin(), nodes[i].end(), cmp);
            for(int j = 0; j < (int) nodes[i].size() - 1; j++){
                int x = nodes[i][j], y = nodes[i][j + 1];
                if(head[x] == head[y]) distinct[lca(x, y)]--;
                distinct[x]++;
            }
            distinct[nodes[i][(int) nodes[i].size() - 1]]++;
        }
    }
    for(int i = root; i >= 1; i--){
        if(!deg[i]) dfs_find(i, i);
    }
}

void solve(int u, int v, int k){
    if((head[u] != head[v]) || (distinct[head[u]] < k)){
        printf("-1\n");
        return;
    }
    int lc = lca(u, v);
    if(distinct[lc] >= k){
        printf("%d\n", val[lc]);
        return;
    }
    int node = lc;
    for(int i = LN - 1; i >= 0; i--){
        if(distinct[dp[node][i]] < k){
            node = dp[node][i];
        }
    }
    node = dp[node][0];
    printf("%d\n", val[node]);
}

int main(){
    scanf("%d %d %d", &n, &m, &q);
    for(int i = 1; i <= n; i++){
        scanf("%d", t + i);
        values.push_back(t[i]);
    }
    sort(values.begin(), values.end());
    values.resize(unique(values.begin(), values.end()) - values.begin());
    for(int i = 1; i <= n; i++){
        t[i] = lower_bound(values.begin(), values.end(), t[i]) - values.begin() + 1;
        nodes[t[i]].push_back(i);
    }
    for(int i = 1; i <= m; i++){
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].w);
    }
    build();
    for(int i = 1; i <= q; i++){
        int u, v, k;
        scanf("%d %d %d", &u, &v, &k);
        solve(u, v, k);
    }
}
```

5.11 Shortest Paths

5.11.1 Count Shortest Paths

```
// Undirected, Weighted Graph without self loops and multiple edges.
// dist[i][j] = Shortest Path from (i) to (j)
// num[i][j]  = Number of Shortest Paths from (i) to (j)

// NOI Social Network (WciPeg)

#include "bits/stdc++.h"
using namespace std;

const int N = 1e2 + 2;
const long long INF = 1e12;

int n, m, u, v, w;
long long dist[N][N], num[N][N];

int main(){
    freopen("ioi.in", "r", stdin);
    scanf("%d %d", &n, &m);
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            dist[i][j] = INF;
        }
        dist[i][i] = 0;
    }
    for(int i = 1; i <= m; i++){
        scanf("%d %d %d", &u, &v, &w);
        dist[u][v] = dist[v][u] = w;
        num[u][v] = num[v][u] = 1;
    }
    for(int k = 1; k <= n; k++){
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                if(dist[i][k] + dist[k][j] < dist[i][j]){
                    dist[i][j] = dist[i][k] + dist[k][j];
                    num[i][j] = num[i][k] * num[k][j];
                }
                else if(dist[i][k] + dist[k][j] == dist[i][j]){
                    num[i][j] += num[i][k] * num[k][j];
                }
            }
        }
    }
    for(int i = 1; i <= n; i++){
        double ans = 0;
        for(int s = 1; s <= n; s++){
            for(int t = 1; t <= n; t++){
                if(s == i || t == i) continue;
                long long numerator = 0, denominator = 0;
                if(dist[s][t] == dist[s][i] + dist[i][t])
                    numerator += (num[s][i] * num[i][t]);
                denominator += (num[s][t]);
                if(denominator != 0) ans += (numerator * 1.0 / denominator);
            }
        }
        printf("%.3f\n", ans);
    }
}
```

5.11.2 Semidynamic Shortest Path Tree

```
// Ans[i] = Shortest Path from S to D when you remove edge (i)
#include <bits/stdc++.h>
using namespace std;

const int MAXE = 50005;
const int MAXN = 7005;
const int INF = (int)(1e9);

// Information about Edges
int U[MAXE], V[MAXE], W[MAXE], VAL[MAXE], E[MAXE];

// Information about Nodes
int DIST[2][MAXN], LEV[MAXN], DSU[MAXN], PAR[MAXN], F[MAXN];

// ANS[i] = Shortest Path when you remove edge number (i)
int ANS[MAXE];

map < int , int > ID[MAXN]; // MAP[u][v] = Edge number of the edge (u->v)
vector < int > edgeList[MAXN]; // List of Edges
vector < int > tree[MAXN]; // Stores any Shortest Path Tree

int N, M, Q, S, D;

bool coolEdge[MAXE];
bool coolNode[MAXN];

inline void dijkstra(int src, int idx){

    for(int i = 0 ; i < N ; i++) DIST[idx][i] = INF;

    set < pair < int , pair < int , int > > > nodes;
    nodes.insert( make_pair(0 , make_pair(src, 0) ) );

    while(!nodes.empty()){

        int u  = (*nodes.begin()).second.first;
        int edgeNo = (*nodes.begin()).second.second;
        int v  = (U[edgeNo] + V[edgeNo]) - (u);
        int c  = (*nodes.begin()).first;

        nodes.erase(nodes.begin());

        if( DIST[idx][u] == INF ){

            // Add to Shortest Path Tree
            if( (idx == 0) and (edgeNo > 0) ){
                coolEdge[edgeNo] = true;
                tree[u].push_back(edgeNo);
                tree[v].push_back(edgeNo);
            }

            DIST[idx][u] = c;
            for(int i = 0 ; i < edgeList[u].size() ; i++){
                edgeNo = edgeList[u][i];
                v = (U[edgeNo] + V[edgeNo]) - (u);
                nodes.insert(make_pair(W[edgeNo] + c, make_pair(v, edgeNo)));
            }
        }
    }

}

bool dfs(int u, int p){

    bool inPath = (u == D);

    for(int i = 0 ; i < tree[u].size() ; i++){

        int edgeNo = tree[u][i];
        int v = (U[edgeNo] + V[edgeNo]) - (u);
        if(edgeNo == p) continue;

        DSU[v] = PAR[v] = u;
        LEV[v] = LEV[u] + 1;

        inPath |= dfs(v, edgeNo);
    }

    if(p != -1) coolEdge[p] = inPath;

    coolNode[u] = inPath;
    return inPath;
}

bool cmp(int x , int y){
    return (VAL[x]) < (VAL[y]);
}

int find(int x){
    if(coolNode[x]) return x;
    return DSU[x] = find(DSU[x]);
}

int goUP(int u, int lca, int val){

    if(LEV[u] <= LEV[lca]) return u;
    if(!coolNode[u]) return DSU[u] = goUP(DSU[u], lca, val); // Visit each edge once

    coolNode[u] = false;
    // Mark this node on Shortest Path as processed

    int p = PAR[u];
    int edgeNo = ID[min(u,p)][max(u,p)];
    ANS[edgeNo] = val;

    return DSU[u] = goUP(DSU[u], lca, val); // Compress Tree
}

int main() {
```

```
cin.tie(0), ios::sync_with_stdio(false);

cin >> N >> M >> Q;
for(int i = 1 ; i <= M ; i++){

    cin >> U[i] >> V[i] >> W[i];
    if(U[i] > V[i]) swap(U[i], V[i]);

    edgeList[U[i]].push_back(i);
    edgeList[V[i]].push_back(i);
    ID[U[i]][V[i]] = i;

    ANS[i] = INF;
}

S = 0, D = N - 1;

dijkstra(S, 0); // Dijkstra from Source
dijkstra(D, 1); // Dijkstra from Destination

int elen = 0;

// Cool edges are those that are in the Shortest Path Tree
for(int i = 1 ; i <= M ; i++){
    if(coolEdge[i]) continue;
    // If edge is in the Shortest Path Tree, Ignore!
    E[ elen++ ] = i;
    VAL[i] = min(DIST[0][U[i]] + DIST[1][V[i]], DIST[0][V[i]] + DIST[1][U[i]]) + W[i];
}


sort(E, E + elen, cmp);
dfs(S, -1);

// Now Cool edges are those which lie on the Shortest Path from S -> D
// Cool nodes are those which lie on the Shortest Path from S -> D

for(int i = 1 ; i <= M ; i++){
    if(!coolEdge[i])  // Ans for all these edges is = Shortest Path from S -> D
        ANS[i] = DIST[0][D];
}

DSU[S] = PAR[S] = S;

for(int i = 0 ; i < N ; i++){
    if(!coolNode[i]){
        // If (i) isn't in Shortest Path
        // Find the first ancestor of (i) which is in Shortest Path
        DSU[i] = find(DSU[i]);
        F[i] = DSU[i];
    }
    else F[i] = i;
}

for(int i = 0 ; i < elen ; i++){

    int edgeNo = E[i];
    int u = U[edgeNo], v = V[edgeNo], w = VAL[edgeNo];

    u = F[u], v = F[v];
    int lca = (LEV[u] < LEV[v]) ? u : v;
```

```
        goUP(u, lca, w);
        goUP(v, lca, w);
    }

    while(Q--){
        int edge;
        cin >> edge;
        edge++;
        if(ANS[edge] == INF) cout << "-1" << '\n';
        else cout << ANS[edge] << '\n';
    }

    return 0;
}
```

5.12 Strongly Connected Components

5.12.1 SCC

```
/*
   ------------ Strongly Connected Components ----------

   This implementation computes the SCCs of a directed graph in linear time.
   Things to set up before using this code:

   - adj[] should be an adjacency list corresponding to the directed graph
   we want to operate on.
   - MAXN should be the maximum number of nodes in the graph. These are 0 indexed.

   kosaraju() computes a vector of vectors, denoting the partition of the graph
   into SCCs. The number of SCCs can be found by looking at scc.size(), and each
   scc is stored in the same vector, indexed from [0, scc.size())

   -----------------------------------------------------
 */

const int MAXN = 100000;

vector < int > adj[MAXN], rev[MAXN];
vector < int > g[MAXN];
vector < bool > visit(MAXN);
vector < vector < int > > scc;

inline void dfs(vector<int> &res, int u) {
    visit[u] = true;
    for (int j = 0; j < (int) g[u].size(); j++) {
        if (!visit[g[u][j]]) {
            dfs(res, g[u][j]);
        }
    }
    res.push_back(u);
}

inline void kosaraju(int nodes) {
    fill(visit.begin(), visit.end(), false);
    vector < int > order;
    for (int i = 0; i < MAXN; i++) {
        g[i] = adj[i];
    }
    for (int i = 0; i < nodes; i++) {
        rev[i].clear();
        if (!visit[i]) {
            dfs(order, i);
        }
    }
    reverse(order.begin(), order.end());
    fill(visit.begin(), visit.end(), false);
    for (int i = 0; i < nodes; i++) {
        for (int j = 0; j < (int) adj[i].size(); j++) {
            rev[adj[i][j]].push_back(i);
        }
    }
    scc.clear();
    for (int i = 0; i < MAXN; i++) {
        g[i] = rev[i];
    }
    for (int i = 0; i < (int) order.size(); i++) {
        if (visit[order[i]]) {
            continue;
        }
        vector < int > component;
        dfs(component, order[i]);
        scc.push_back(component);
    }
}
// End of Strongly Connected Components
```

6 Maths and DP

6.1 Bitmask DP

6.1.1 TSP

```
/*
   Given a weighted graph, determine a cycle of minimum total distance which visits
   each node exactly once and returns to the starting node. This is known as the
   traveling salesman problem (TSP). Since this implementation uses bitmasks with
   32-bit integers, the maximum number of nodes must be less than 32.
   shortest_hamiltonian_cycle() applies to a global adjacency matrix adj[][], which
   must be populated with add_edge() before the function call.
   Time Complexity:
   - O(2^n * n^2) per call to shortest_hamiltonian_cycle(), where n is the number
   of nodes.
   Space Complexity:
   - O(n^2) for storage of the graph, where n is the number of nodes.
   - O(2^n * n^2) auxiliary heap space for shortest_hamiltonian_cycle().
 */

#include <algorithm>

const int MAXN = 20, INF = 0x3f3f3f3f;
int adj[MAXN][MAXN], dp[1 << MAXN][MAXN], order[MAXN];

void add_edge(int u, int v, int w) {
    adj[u][v] = w;
    adj[v][u] = w;  // Remove this line if the graph is directed.
}
```

```
int shortest_hamiltonian_cycle(int nodes) {
    int max_mask = (1 << nodes) - 1;
    for (int i = 0; i <= max_mask; i++) {
        std::fill(dp[i], dp[i] + nodes, INF);
    }
    dp[1][0] = 0;
    for (int mask = 1; mask <= max_mask; mask += 2) {
        for (int i = 1; i < nodes; i++) {
            if ((mask & 1 << i) != 0) {
                for (int j = 0; j < nodes; j++) {
                    if ((mask & 1 << j) != 0) {
                        dp[mask][i] = std::min(dp[mask][i],
                                dp[mask ^ (1 << i)][j] + adj[j][i]);
                    }
                }
            }
        }
    }
    int res = INF + INF;
    for (int i = 1; i < nodes; i++) {
        res = std::min(res, dp[max_mask][i] + adj[i][0]);
    }
    int mask = max_mask, old = 0;
    for (int i = nodes - 1; i >= 1; i--) {
        int bj = -1;
        for (int j = 1; j < nodes; j++) {
            if ((mask & 1 << j) != 0 && (bj == -1 ||
                        dp[mask][bj] + adj[bj][old] > dp[mask][j] + adj[j][old])) {
                bj = j;
            }
        }
        order[i] = bj;
        mask ^= 1 << bj;
        old = bj;
    }
    return res;
}

/*** Example Usage and Output:
  The shortest hamiltonian cycle has length 5.
  Take the path: 0->3->2->4->1->0.
 ***/

#include <iostream>
using namespace std;

int main() {
    int nodes = 5;
    add_edge(0, 1, 1);
    add_edge(0, 2, 10);
    add_edge(0, 3, 1);
    add_edge(0, 4, 10);
    add_edge(1, 2, 10);
    add_edge(1, 3, 10);
    add_edge(1, 4, 1);
    add_edge(2, 3, 1);
    add_edge(2, 4, 1);
    add_edge(3, 4, 10);
    cout << "The shortest hamiltonian cycle has length "
        << shortest_hamiltonian_cycle(nodes) << "." << endl
        << "Take the path: ";
    for (int i = 0; i < nodes; i++) {
        cout << order[i] << "->";
    }
    cout << order[0] << "." << endl;
    return 0;
}
```

6.2 CRT

6.2.1 Chinese Remainder Theorem

```
// SPOJ FACTMUL
#include "bits/stdc++.h"
using namespace std;

const long long MOD = 109546051211;
const int P = 186583;
const int Q = 587117;

// Chinese Remainder Theorem: O(n^2 log n)
// All credit for this code goes to Alex Li.
inline long long perform_mod(long long a, long long m) {
    long long r = a % m;
    return (r >= 0) ? r : (r + m);
}

inline long long mod_inverse(long long a, long long m) {
    a = perform_mod(a, m);
    if (a == 0) {
        return 0;
    } else {
        return perform_mod((1 - m * mod_inverse(m % a, a)) / a, m);
    }
}

// n is the size of a and p
// (a[i], p[i]) denotes the remainder of the value we're trying to compute modulo p[i].
// all elements in p[] should be pairwise coprime.
inline long long garner_restore(int n, int a[], int p[]) {
    vector < long long > x(a, a + n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            x[i] = mod_inverse((long long) p[j], (long long) p[i]) * (x[i] - x[j]);
        }
        x[i] = (x[i] % p[i] + p[i]) % p[i];
    }
    long long res = x[0], m = 1;
    for (int i = 1; i < n; i++) {
        m *= p[i - 1];
        res += x[i] * m;
    }
    return res;
}
// End of Template

inline int get(int n, int modulo) {
    long long result = 1;
    long long factorial = 1;
    for (int i = 1; i <= n; i++) {
        factorial = (factorial * i) % modulo;
        result = (result * factorial) % modulo;
    }
    return result;
}

int main() {
    int n; cin >> n;
    int a[2], p[2];
    a[0] = get(n, P); p[0] = P;
    a[1] = get(n, Q); p[1] = Q;
    cout << garner_restore(2, a, p) << endl;
}
```

6.3 Combinatorics

6.3.1 Combinatorics

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

// Returns N choose R. Takes O(N log MOD) for precomputation, and then O(1) per query.
// Adjust N and MOD as required -> MOD must be prime.
// Call preprocess() from your main() once.
// After that, choose(n, k) returns (n choose k) modulo (MOD).

const int N = 200005;
const int MOD = 1000000007;

int fact[N], inv_fact[N];

inline int prod(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD) res %= MOD;
    return res;
}

inline int add(int x, int y){
    int res = x + y;
    if (res < 0) res += MOD;
    if(res >= MOD) res -= MOD;
    return res;
}

inline int power(int x, int y){
    if(y == 0) return 1;
    int res = power(x, y >> 1);
    res = prod(res, res);
    if(y & 1) res = prod(res, x);
    return res;
}

inline void preprocess(){
    fact[0] = inv_fact[0] = 1;
    for(int i = 1; i < N; i++) fact[i] = prod(fact[i - 1], i);
    for(int i = N - 1; i >= 1; i--) inv_fact[i] = power(fact[i], MOD - 2);
}

inline int choose(int n, int r){
    if(r < 0 || r > n) return 0;
    return prod(fact[n], prod(inv_fact[r], inv_fact[n - r]));
}

// End of Counting Template

int main(){
    preprocess();
    cout << choose(10, 2) << endl;
}
```

6.3.2 Equivalence Class DP

```
// Codeforces - On the bench

#include "bits/stdc++.h"
using namespace std;

const int MOD = 1000000007;

inline int prod(int x, int y) {
    long long res = x * 1LL * y;
    if (res >= MOD) {
        res %= MOD;
    }
    return res;
}

inline int add(int x, int y) {
    int res = x + y;
    if (res < 0) {
        res += MOD;
    }
    if (res >= MOD) {
        res -= MOD;
    }
    return res;
}

const int N = 333;

int n;
int arr[N];
int c[N];
vector < int > values;

inline int normalize(int x) {
    int z = 0;
    for (int i = 2; i * i <= x; i++) {
        int t = 0;
        while (x % i == 0) {
            x /= i;
            t += 1;
        }
        if (t & 1) {
            if (z == 0) z = 1;
            z *= i;
        }
    }
    if (x != 1) {
        if (z == 0) z = 1;
        z *= x;
    }
    return z;
}

inline void compress() {
    sort(values.begin(), values.end());
    for (int i = 1; i <= n; i++) {
        arr[i] = lower_bound(values.begin(), values.end(), arr[i]) - values.begin() + 1;
    }
}

/**
  A more general problem:

  Given a list of G types of balls where:
  1) Balls of the same type are indistinguishable, and
  2) We have g_i balls of type i for all 1 <= i <= G.

  Compute the number of ways to arrange all balls in a line such that
  no two adjacent balls are of the same type (we define such a pair
  as a bad pair). The below code returns this answer modulo 1000000007.
 **/

const int MAX = 333;

int fact[MAX];
int choose[MAX][MAX];
int dp[MAX][MAX];

inline void precompute() {
    fact[0] = 1;
    for (int i = 1; i < N; i++) {
        fact[i] = prod(fact[i - 1], i);
    }
    choose[0][0] = 1;
    for (int i = 1; i < N; i++) {
        choose[i][0] = choose[i][i] = 1;
        for (int j = 1; j < i; j++) {
            choose[i][j] = add(choose[i - 1][j], choose[i - 1][j - 1]);
        }
    }
}

inline int ncr(int n, int r) {
    if (r < 0 || r > n) {
        return 0;
    }
    return choose[n][r];
}

inline int solve(vector < int > group_sizes) {
    precompute();
    int g = (int) group_sizes.size();
    int total = 0;
    for (int s: group_sizes) {
        total += s;
    }
    assert(total < MAX);
    group_sizes.insert(group_sizes.begin(), 0); // Just to make stuff 1-indexed.

    // dp[i][x] = Number of permutations using first (i) groups of balls
    // that have exactly (x) bad pairs of neighbours.
    memset(dp, 0, sizeof dp);

    // Base Case with just one group.
    dp[1][group_sizes[1] - 1] = 1;

    int placed_till_now = 0, gaps_till_now = 0;
    for (int i = 1; i < g; i++) { // Augment dp[i + 1][...] states using dp[i][...] states
        placed_till_now += group_sizes[i];
        gaps_till_now = placed_till_now - 1; // number of "holes" we can place stuff at, excluding the boundaries.
        for (int x = 0; x <= gaps_till_now; x++) { // Iterating over bad pairs which must be <= gaps_till now
            for (int p = 1; p <= min(group_sizes[i + 1], gaps_till_now + 2); p++) { // Number of parts we break the
    (i + 1)th group into.
                for (int b = 0; b <= min(x + 1, p); b++) { // Number of bad pairs we eliminate.
                    // We had x bad pairs, we eliminated b of them, but we added some more because of the blocks
                    // leaving us with exactly z bad pairs.
                    int z = x - b + (group_sizes[i + 1] - p);
                    // Let's compute number of ways to achieve z bad pairs. First, we divide
                    // the (i + 1)th group we are looking at into p nonempty parts.
                    // This formula considers ordering within the parts as well, so no need to multiply by p!
                    int ways_to_divide_group = ncr(group_sizes[i + 1] - 1, p - 1);

                    // From all permutations that we are currently looking at, we will:
                    int num_permutations = dp[i][x]; // 1) pick a permutation
                    int num_bad_slots = ncr(x, b); // 2) pick b bad slots to eliminate from that permutation.
                    int num_good_slots = ncr(gaps_till_now + 2 - x, p - b); // 3) pick good slots to place remaining
    parts.

                    int ways_to_place = prod(num_permutations, prod(num_bad_slots, num_good_slots)); // Product of
    above three

                    // Finally, we merge the two quanitities we computed to get total number of ways
                    int total_ways = prod(ways_to_divide_group, ways_to_place);

                    // Augment dp state
                    dp[i + 1][z] = add(dp[i + 1][z], total_ways);
                }
            }
        }
    }
    // Use all balls and have 0 bad pairs.
    return dp[g][0];
}

// End of Generic Template

int main() {
    ios :: sync_with_stdio(false);
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> arr[i];
        arr[i] = normalize(arr[i]);
        values.push_back(arr[i]);
    }
    compress();
    for (int i = 1; i <= n; i++) {
        c[arr[i]] += 1;
    }
    vector < int > class_sizes;
    for (int i = 1; i <= n; i++) {
        if (c[i] > 0) {
            class_sizes.push_back(c[i]);
        }
    }
    int ans = solve(class_sizes);
    // In this problem, objects of the same type are distinguishable
    // so we multiply answer by factorials of group sizes.
    for (int i = 1; i <= n; i++) {
        ans = prod(ans, fact[c[i]]);
    }
    cout << ans << endl;
}
```

6.3.3 Gerald and Giant Chess

```
// Given a HxW grid with N black cells, compute number of paths from (1, 1) to (H, W)
// that does not go through any black cell. Report the answer modulo 1000000007.
// H, W <= 100000 and N <= 5000.
// This solution is O(N^2).

#include "bits/stdc++.h"
using namespace std;

const int H = 200005;
const int N = 2005;
const int MOD = 1000000007;

int h, w, n;
int fact[H], inv_fact[H];
int dp[N];

inline int prod(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD) res %= MOD;
    return res;
}

inline int add(int x, int y){
    int res = x + y;
    if (res < 0) res += MOD;
    if(res >= MOD) res -= MOD;
    return res;
}

inline int power(int x, int y){
    if(y == 0) return 1;
    int res = power(x, y >> 1);
    res = prod(res, res);
    if(y & 1) res = prod(res, x);
    return res;
}

inline void preprocess(){
    fact[0] = inv_fact[0] = 1;
    for(int i = 1; i < H; i++) fact[i] = prod(fact[i - 1], i);
    for(int i = H - 1; i >= 1; i--) inv_fact[i] = power(fact[i], MOD - 2);
}

inline int choose(int n, int r){
    if(r < 0 || r > n) return 0;
    return prod(fact[n], prod(inv_fact[r], inv_fact[n - r]));
}
```

```
inline int solve(int a1, int b1, int a2, int b2) {
    int length_of_path = (a2 - a1) + (b2 - b1);
    int number_of_turns = (b2 - b1);
    return choose(length_of_path, number_of_turns);
}

int main() {
    ios :: sync_with_stdio(false);
    preprocess();
    cin >> h >> w >> n;
    vector < pair < int, int > > black_cells;
    for (int i = 1; i <= n; i++) {
        int foo, bar; cin >> foo >> bar;
        black_cells.push_back({foo, bar});
    }
    black_cells.push_back({h, w}); // For convenience.
    sort(black_cells.begin(), black_cells.end());
    // dp[i] = number of paths from (1, 1) to (x_i, y_i)
    // which doesn't pass through any (x_j, y_j) for all j < i.
    // Standard counting trick: Fix the first time we break the rule.
    for (int i = 0; i <= n; i++) {
        int x = black_cells[i].first;
        int y = black_cells[i].second;
        dp[i] = solve(1, 1, x, y);
        for (int j = 0; j < i; j++) {
            int a = black_cells[j].first;
            int b = black_cells[j].second;
            if (a <= x and b <= y) {
                dp[i] = add(dp[i], -prod(dp[j], solve(a, b, x, y)));
            }
        }
    }
    cout << dp[n] << endl;
}
```

6.3.4 Lucas Theorem

```
// Implementation of Lucas's theorem that computes Binomial(n, k) mod p, for
// a prime p, in terms of the base p expansions of n and k. This can be used
// with the Chinese remainder theorem to compute Binomial(n, k) mod m, for some
// square-free integer m.

#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

ll p;
vector<ll> factorials, inverse_factorials;

ll powmod(ll b, ll e) {
    ll ans = 1;
    while (e) {
        if (e % 2) ans = (ans * b) % p;
        b = (b * b) % p;
        e /= 2;
    }
    return ans;
}

void GetFactorials() {
    factorials.resize(p);
    inverse_factorials.resize(p);
    factorials[0] = 1;
    inverse_factorials[0] = 1;
    for (int i = 1; i < p; i++) {
        factorials[i] = (i * factorials[i - 1]) % p;
        inverse_factorials[i] = powmod(factorials[i], p - 2);
    }
}

ll Binomial(ll n, ll k) {
    if (n < 0 || k < 0 || k > n) return 0;
    ll tmp = (factorials[n] * inverse_factorials[k]) % p;
    return (tmp * inverse_factorials[n - k]) % p;
}

ll LucasBinomial(ll n, ll k) {
    ll ans = 1;
    while (n || k) {
        ans = (ans * Binomial(n % p, k % p)) % p;
        n /= p;
        k /= p;
    }
    return ans;
}

int main() {
    p = 10000019; // Set the prime as required
    GetFactorials(); // Precompute Factorials
    // Now LucasBinomial(n, k) returns (n choose k) modulo p.
    assert(LucasBinomial(1000000, 500000) == 2421826);
    return 0;
}
```

6.4 DP Optimizations

6.4.1 Divide and Conquer

```
// Codeforces VK Cup 2016 - Divide and Conquer DP Optimization
// This can be used when best[i][j] <= best[i + 1][j], where best[i][j] is the index (k) which maximises dp[i][j]
// You can exploit the monotonicity of the best[][] array to speed up your code.
#include <bits/stdc++.h>
using namespace std;

const int N = 200005;
const int K = 55;

int n, k, t[N];
double t_sum[N], t_inv[N], pre[N], dp[K][N];

/*
   dp[j][i]   = min(dp[j - 1][k] + cost[k + 1][i]), where (1 <= k < i).
   cost[i][j] = (t[i]) / (t[i]) +
   (t[i] + t[i + 1]) / (t[i + 1]) +
   (t[i] + t[i + 1] + t[i + 2]) / (t[i + 2]) + ....
   (t[i] + t[i + 1].... + t[j]) / (t[j])
   = (pre[j] - pre[i - 1]) - t_sum[i - 1] * (t_inv[j] - t_inv[i - 1])
 */

inline double cost(int i, int j){
    return (pre[j] - pre[i - 1]) - t_sum[i - 1] * (t_inv[j] - t_inv[i - 1]);
}

inline void compute(int j, int l, int r, int qs, int qe){
    int i = (l + r) >> 1, best_idx = -1;
    dp[j][i] = 1e18;
    for(int k = qs; k <= min(qe, i - 1); k++){
        if(dp[j - 1][k] + cost(k + 1, i) < dp[j][i]){
            dp[j][i] = dp[j - 1][k] + cost(k + 1, i);
            best_idx = k;
        }
    }
    if(i > l) compute(j, l, i - 1, qs, best_idx);
    if(i < r) compute(j, i + 1, r, best_idx, qe);
}

int main(){
    scanf("%d %d", &n, &k);
    for(int i = 1; i <= n; i++){
        scanf("%d", t + i);
        t_sum[i] = t_sum[i - 1] + t[i];
        t_inv[i] = t_inv[i - 1] + (1.0 / t[i]);
        pre[i]   = pre[i - 1] + (t_sum[i] / t[i]);
    }
    for(int i = 1; i <= n; i++) dp[1][i] = dp[1][i - 1] + (pre[i] - pre[i - 1]);
    for(int i = 2; i <= k; i++) compute(i, 1, n, 1, n);
    printf("%.10f\n", dp[k][n]);
}
```

6.4.2 Semidynamic Convex Hull Trick

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

const int N = 3e5 + 5;


/** ----- Minimum Convex Hull Trick Template ------- */

struct cht{
    struct line{
        long long a , b;
        double xleft;
        bool type;
        line(long long _a , long long _b){
            a = _a;
            b = _b;
            type = 0;
        }
        bool operator < (const line &other) const{
            if(other.type){
                return xleft < other.xleft;
            }
            return a > other.a;
        }
    };
    inline double intersect(line x , line y){
        return 1.0 * (y.b - x.b) / (x.a - y.a);
    }
    multiset < line > hull;
```

```
cht(){
    hull.clear();
}
typedef set < line > :: iterator iter;
inline bool has_left(iter node){
    return node != hull.begin();
}
inline bool has_right(iter node){
    return node != prev(hull.end());
}
inline void update_border(iter node){
    if(has_right(node)){
        line temp = *next(node);
        hull.erase(hull.find(temp));
        temp.xleft = intersect(*node, temp);
        hull.insert(temp);
    }
    if(has_left(node)){
        line temp = *node;
        temp.xleft = intersect(*prev(node), temp);
        hull.erase(node);
        hull.insert(temp);
    }
    else{
        line temp = *node;
        hull.erase(node);
        temp.xleft = -1e18;
        hull.insert(temp);
    }
}
inline bool useless(line left , line middle , line right){
    return intersect(left, middle) > intersect(middle, right);
}
inline bool useless(iter node){
    if(has_left(node) && has_right(node)){
        return useless(*prev(node), *node, *next(node));
    }
    return 0;
}
// add line with equation y = (a * x + b)
inline void add(long long a , long long b){
    line temp = line(a, b);
    auto it = hull.lower_bound(temp);
    if(it != hull.end() && it -> a == a){
        if(it -> b > b){
            hull.erase(it);
        }
        else{
            return;
        }
    }
    hull.insert(temp);
    it = hull.find(temp);
    if(useless(it)){
        hull.erase(it);
        return;
    }
    while(has_left(it) && useless(prev(it))){
        hull.erase(prev(it));
    }
    while(has_right(it) && useless(next(it))){
        hull.erase(next(it));
        }
        update_border(it);
    }
    // get minimum value of (m * x + c) for given x
    inline long long query(long long x){
        if(hull.empty()){
            return 5e18;
        }
        line query(0, 0);
        query.xleft = x;
        query.type = 1;
        auto it = hull.lower_bound(query);
        it = prev(it);
        return it -> a * x + it -> b;
    }
};


/** ----------- End of Template ------------ **/


cht tree[N * 4];

inline void update_tree(int node, int l, int r, int qs, int qe, int m, int c){
    if(l > qe || r < qs) return;
    if(l >= qs && r <= qe){
        tree[node].add(-m, -c);
        return;
    }
    int mid = (l + r) >> 1;
    update_tree(node * 2, l, mid, qs, qe, m, c);
    update_tree(node * 2 + 1, mid + 1, r, qs, qe, m, c);
}

inline long long query_tree(int node, int l, int r, int pos, int x){
    long long cur = -tree[node].query(x);
    if(l != r && pos){
        int mid = (l + r) >> 1;
        if(mid >= pos) cur = max(cur, query_tree(node * 2, l, mid, pos, x));
        else cur = max(cur, query_tree(node * 2 + 1, mid + 1, r, pos, x));
    }
    return cur;
}

int t, q, x, id;
pair < int, int > query[N], range[N];

int main(){
    scanf("%d", &q);
    for(int i = 1; i <= q; i++){
        scanf("%d", &t);
        if(t == 1){
            scanf("%d %d", &query[i].first, &query[i].second);
            range[i] = {i, q};
        }
        else if(t == 2){
            scanf("%d", &id);
            range[id] = {range[id].first, i - 1};
        }
        else{
            scanf("%d", &x);
            range[i] = {x, q + 1};
        }
    }
    for(int i = 1; i <= q; i++){
        if(range[i].first == 0 && range[i].second == 0) continue;
        if(range[i].second <= q){
            update_tree(1, 1, q, range[i].first, range[i].second, query[i].first, query[i].second);
        }
        else{
            long long res = query_tree(1, 1, q, i, range[i].first);
            if(res == -5e18) puts("EMPTY SET");
            else printf("%lld\n", res);
        }
    }
}
```

6.5 FFT

6.5.1 FFT

```
// Convolution using the fast Fourier transform (FFT).
//
// INPUT:
//     a[1...n]
//     b[1...m]
//
// OUTPUT:
//     c[1...n+m-1] such that c[k] = sum_{i=0}^k a[i] b[k-i]
//
// Alternatively, you can use the DFT() routine directly, which will
// zero-pad your input to the next largest power of 2 and compute the
// DFT or inverse DFT.

#include <iostream>
#include <vector>
#include <complex>

using namespace std;

typedef long double DOUBLE;
typedef complex<DOUBLE> COMPLEX;
typedef vector<DOUBLE> VD;
typedef vector<COMPLEX> VC;

struct FFT {
    VC A;
    int n, L;

    int ReverseBits(int k) {
        int ret = 0;
        for (int i = 0; i < L; i++) {
            ret = (ret << 1) | (k & 1);
            k >>= 1;
        }
        return ret;
    }

    void BitReverseCopy(VC a) {
        for (n = 1, L = 0; n < a.size(); n <<= 1, L++) ;
        A.resize(n);
        for (int k = 0; k < n; k++)
```

```
            A[ReverseBits(k)] = a[k];
    }

    VC DFT(VC a, bool inverse) {
        BitReverseCopy(a);
        for (int s = 1; s <= L; s++) {
            int m = 1 << s;
            COMPLEX wm = exp(COMPLEX(0, 2.0 * M_PI / m));
            if (inverse) wm = COMPLEX(1, 0) / wm;
            for (int k = 0; k < n; k += m) {
                COMPLEX w = 1;
                for (int j = 0; j < m/2; j++) {
                    COMPLEX t = w * A[k + j + m/2];
                    COMPLEX u = A[k + j];
                    A[k + j] = u + t;
                    A[k + j + m/2] = u - t;
                    w = w * wm;
                }
            }
        }
        if (inverse) for (int i = 0; i < n; i++) A[i] /= n;
        return A;
    }

    // c[k] = sum_{i=0}^k a[i] b[k-i]
    VD Convolution(VD a, VD b) {
        int L = 1;
        while ((1 << L) < a.size()) L++;
        while ((1 << L) < b.size()) L++;
        int n = 1 << (L+1);

        VC aa, bb;
        for (size_t i = 0; i < n; i++) aa.push_back(i < a.size() ? COMPLEX(a[i], 0) : 0);
        for (size_t i = 0; i < n; i++) bb.push_back(i < b.size() ? COMPLEX(b[i], 0) : 0);

        VC AA = DFT(aa, false);
        VC BB = DFT(bb, false);
        VC CC;
        for (size_t i = 0; i < AA.size(); i++) CC.push_back(AA[i] * BB[i]);
        VC cc = DFT(CC, true);

        VD c;
        for (int i = 0; i < a.size() + b.size() - 1; i++) c.push_back(cc[i].real());
        return c;
    }

};

int main() {
    double a[] = {1, 3, 4, 5, 7};
    double b[] = {2, 4, 6};

    FFT fft;
    VD c = fft.Convolution(VD(a, a + 5), VD(b, b + 3));

    // expected output: 2 10 26 44 58 58 42
    for (int i = 0; i < c.size(); i++) cerr << c[i] << " ";
    cerr << endl;

    return 0;
}
```

6.5.2 NTT

```
// SPOJ MAXMATCH
#include "bits/stdc++.h"
using namespace std;

/*
   ========== Number Theoretic Transform ===========

   The code below can be used to multiply two polynomials in O(n log n).
   The multiplication happens modulo 5 * (2 ^ 25) + 1 which is around 1.6e8.
   This implementation has been thoroughly tested and can be used when
   dealing with integers. Some implementation details:

   - n should be a power of 2. Generally, if we are multiplying two degree 'd'
   polynomials, n should be the smallest power of two greater than 2 * d.
   - Changing the modulo requires some number theoretic results. Most modulos
   do not work. Here is what we do to deal with modulos:
   - Suppose the modulo is M. M must be prime and of the form 2^k * x + 1
   where k >= ceil(log n) and x >= 1. To make the code below work for a particular
   M, we need to change the 4 constants below. Here is how we find the
constants:
- mod: M
- root_pw: 2^k
- root: Let p = find_primitive_root(M). Then root = (p ^ x) % M
- root_1: inverse(root, M)

We can work out these values offline by using the find_primitive_root()
and inverse() functions below, and then hardcode them into the program.

- Suppose the modulo is not of the form 2^k * x + 1, and we know that in
the product polynomial, the coefficients will be less than ~1e15. Then
we can compute NTT using large primes p1, p2 (around 1e7) and then
compute the value modulo (p1 * p2) using CRT. This will give us a
very high precision result :)

- P1 =  5 * (2 ^ 25) + 1
P2 =  7 * (2 ^ 20) + 1
P1 * P2 >= 1e15.
==================================================
 */

inline long long gcd(long long a, long long b, long long &s, long long &t) {
    if (b == 0) {
        t = 0;
        s = (a < 0) ? -1 : 1;
        return (a < 0) ? -a : a;
    } else {
        long long g = gcd(b, a % b, t, s);
        t -= a / b * s;
        return g;
    }
}

inline long long inverse(long long n, long long mod) {
    long long s, t;
    gcd(n, mod, s, t);
    return (s > 0) ? s : s + mod;
}

const long long mod = 5 * (1 << 25) + 1;
```

```
long long root = 243;
long long root_1 = 114609789;
const long long root_pw = 1 << 25;

inline void fft (vector < long long > & a, bool invert) {
    int n = (int) a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j >= bit; bit >>= 1) {
            j -= bit;
        }
        j += bit;
        if (i < j) {
            swap(a[i], a[j]);
        }
    }
    for (int len = 2; len <= n; len <<= 1) {
        long long wlen = invert ? root_1 : root;
        for (long long i = len; i < root_pw; i <<= 1)
            wlen = (long long) (wlen * 1ll * wlen % mod);
        for (int i = 0; i < n; i += len) {
            long long w = 1;
            for (int j = 0; j < len / 2; j++) {
                long long u = a[i + j];
                long long v = (long long) (a[i + j + len / 2] * 1ll * w % mod);
                a[i + j] = u + v < mod ? u + v : u + v - mod;
                a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + mod;
                w = (long long) (w * 1ll * wlen % mod);
            }
        }
    }
    if (invert) {
        long long nrev = inverse(n, mod);
        for (int i = 0; i < n; i++)
            a[i] = (long long) (a[i] * 1ll * nrev % mod);
    }
}

// ============ Stuff to figure out the 4 constants =============
inline int power(int a, int b, int modulo) {
    if (b == 0) {
        return 1 % modulo;
    }
    if (b == 1) {
        return a % modulo;
    }
    int res = power(a, b / 2, modulo);
    res = (res * 1LL * res) % modulo;
    if (b & 1) {
        res = (res * 1LL * a) % modulo;
    }
    return res;
}

vector < int > factorize(int x) {
    // Returns prime factors of x
    vector < int > primes;
    for (int i = 2; i * i <= x; i++) {
        if (x % i == 0) {
            primes.push_back(i);
            while (x % i == 0) {
                x /= i;
            }
        }
    }
    if (x != 1) {
        primes.push_back(x);
    }
    return primes;
}

inline bool test_primitive_root(int a, int m) {
    // Is 'a' a primitive root of modulus 'm'?
    // m must be of the form 2^k * x + 1
    int exp = m - 1;
    int val = power(a, exp, m);
    if (val != 1) {
        return false;
    }
    vector < int > factors = factorize(exp);
    for (int f: factors) {
        int cur = exp / f;
        val = power(a, cur, m);
        if (val == 1) {
            return false;
        }
    }
    return true;
}

inline int find_primitive_root(int m) {
    // Find primitive root of the modulus 'm'.
    // m must be of the form 2^k * x + 1
    for (int i = 2; ; i++) {
        if (test_primitive_root(i, m)) {
            return i;
        }
    }
}

// End of NTT Template.

const int MAXN = 100000;
const int SIZE = 1 << 19;

int n;
char s[MAXN];

int main() {
    freopen ("inp.in", "r", stdin);
    // Adjust constants
    int primitive_root = find_primitive_root(mod);
    root = power(primitive_root, 5, mod); // 5 = x from the template defn.
    root_1 = inverse(root, mod);
    // Solve problem
    scanf("%s", s);
    n = strlen(s);
    int off = n + 1;
    vector < long long > result(SIZE, 0);
    for (char x = 'a'; x <= 'c'; x++) {
        vector < long long > a(SIZE, 0);
        vector < long long > b(SIZE, 0);
        vector < long long > c(SIZE, 0);
        for (int i = 0; i < n; i++) {
            if (s[i] == x) {
                a[(i + 1) + off] = 1;
                b[-(i + 1) + off] = 1;
            }
        }
        fft(a, false);
        fft(b, false);
        for (int i = 0; i < SIZE; i++) {
            c[i] = a[i] * b[i] % mod;
        }
        fft(c, true);
        for (int i = 0; i < SIZE; i++) {
            result[i] += c[i];
        }
    }
    int mx_val = 0;
    set < int > indices;
    for (int i = 1; i <= n; i++) {
        int j = i + off + off;
        if (result[j] > mx_val) {
            mx_val = result[j];
            indices.clear();
            indices.insert(i);
        } else if (result[j] == mx_val) {
            indices.insert(i);
        }
    }
    printf("%d\n", mx_val);
    for (int i: indices) {
        printf("%d ", i);
    }
    printf("\n");
}
```

6.6 Integration

6.6.1 Revolving Curve

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

const long double PI = acos(-1);
int N, A, B, slices, stacks;
int coef[100], sq_coef[100];
double ind_coef[100];

double power_(double x, int y) {
    double ans = 1;
    for (int i = 0; i < y; i++)
        ans *= x;
    return ans;
}

void square_pol() {
    memset(sq_coef, 0, sizeof(sq_coef));

    for (int i = 0; i <= N; i++) {
        for (int k = 0; k <= N; k++) {
            int c = i + k;
            sq_coef[c] += coef[i] * coef[k];
        }
    }
}

double evaluate(double x) {
    double ans = 0;
    for (int i = 0; i <= N; i++) {
        ans += coef[i] * power_(x, i);
    }
    return ans;
}

double indefinite_evaluate(double x) {
    double ans = 0;
    for (int i = 0; i <= 2*N + 1; i++) {
        ans += ind_coef[i] * power_(x, i);
    }
    return ans;
}

void indefinite_integral() {
    for (int i = 2*N + 1; i > 0; i--) {
        ind_coef[i] = sq_coef[i-1]*1.0 / (i);
    }
}

double definite_integral() {
    square_pol();
    indefinite_integral();
    return PI * (indefinite_evaluate(B) - indefinite_evaluate(A));
}

double approximate() {
    double theta = 2*PI / slices;
    double width = (B - A)* (1.0) / stacks * 1.0;

    double volume = 0;
    double start_ = A, end_ = A + width;
    for (int i = 0; i < stacks; i++) {

        double px1 = evaluate(start_);
        double px2 = evaluate(end_);
        double m = (px2 - px1) / (end_ - start_);
        double c = px2 - m * end_;

        volume += (end_ - start_) * c * c;
        volume += m * c * (end_ * end_ - start_ * start_);
        volume += m * m * (end_*end_*end_ - start_*start_*start_) / 3;

        start_ += width;
        end_ += width;
    }

    return volume * sin(theta) / 2;
}

int main() {

    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        cin >> N;
        for (int i = N; i >= 0; i--) {
            cin >> coef[i];
        }

        cin >> A >> B;
        cin >> slices >> stacks;

        double real_volume = definite_integral();
        double app_volume = approximate() * slices;

        cout << "Case " << t + 1 << ": ";
        cout << fixed << setprecision(4);
        cout << fabs((real_volume - app_volume) / real_volume) * 100.0 << "\n";
    }

    return 0;
}
```

6.6.2 Simpsons Rule

```
/*
   numercial integration.
 */

const double eps = 1e-8;
const double PI = acos(-1);

double f (double x) {
    /* return value of function at value of x */
}

inline double simpson(double fl,double fr,double fmid,double l,double r) { return (fl+fr+4.0*fmid)*(r-l)/6.0; }

double rsimpson(double slr,double fl,double fr,double fmid,double l,double r) {
    double mid = (l+r)*0.5;
    double fml = f((l+mid)*0.5);
    double fmr = f((mid+r)*0.5);
    double slm = simpson(fl,fmid,fml,l,mid);
    double smr = simpson(fmid,fr,fmr,mid,r);
    if(fabs(slr-slm-smr) < eps) return slm+smr;
    return rsimpson(slm,fl,fmid,fml,l,mid)+rsimpson(smr,fmid,fr,fmr,mid,r);
}

double integrate(double l,double r) {
    double mid = (l+r)*.5;
    double fl = f(l);
    double fr = f(r);
    double fmid = f(mid);
    return rsimpson(simpson(fl,fr,fmid,l,r),fl,fr,fmid,l,r);
}
```

6.7 Linear Algebra

6.7.1 Basis

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting.  This can be used for computing
// the rank of a matrix.
//
```

```
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxm matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

//           To get basis, we store the indices of the columns in the
//           original matrix.

#include <bits/stdc++.h>
using namespace std;
const double EPSILON = 1e-10;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

bool is_linear_combination(const VVT& A, const vector<int> indx, VT v) {
    for (int i = 0; i < v.size(); i++) {
        if (!v[i]) continue;
        bool good = false;
        for (int k : indx) {
            if (A[i][k]) {
                good = true;
                break;
            }
        }
        if (!good) return false;
    }
    return true;
}

int get_basis(VVT &a, vector<int>& basis_indx) {
    int n = a.size();
    int m = a[0].size();
    int r = 0;
    for (int c = 0; c < m && r < n; c++) {
        int j = r;
        for (int i = r + 1; i < n; i++)
            if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
        if (fabs(a[j][c]) < EPSILON) continue;
        swap(a[j], a[r]);

        T s = 1.0 / a[r][c];
        for (int j = 0; j < m; j++) a[r][j] *= s;
        for (int i = 0; i < n; i++) if (i != r) {
            T t = a[i][c];
            for (int j = 0; j < m; j++) {
                a[i][j] -= t * a[r][j];
            }
        }
        basis_indx.push_back(c);
        r++;
    }
    return r;
}

int main() {

    VVT AA = A;
    vector<int> basis_indx;
    int rank = get_basis(A, basis_indx);
    int free = N - rank;

    if (is_linear_combination(AA, basis_indx, v)) {
        // vector v is a linear combination of columns of matrix AA
    }

    return 0;
}
```

6.7.2 Gaussian with Rationals

```
/*
   Probability Paradox

 */

#include "bits/stdc++.h"
using namespace std;

inline long long gcd(long long a, long long b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

struct rational {
    long long p, q;
    void red() {
        if (q < 0) {
            p *= -1;
            q *= -1;
        }
        long long t = gcd((p >= 0 ? p : -p), q);
        p /= t;
        q /= t;
    }
    rational() {} rational(long long p_): p(p_), q(1)
    {}rational(long long p_, long long q_): p(p_), q(q_) {
                                                            red();
                                                        }
    bool operator==(const rational& rhs) const {
        return p == rhs.p && q == rhs.q;
    }
    bool operator!=(const rational& rhs) const {
        return p != rhs.p || q != rhs.q;
    }
    bool operator<(const rational& rhs) const {
        return p * rhs.q < rhs.p * q;
    }
    bool operator>(const rational& rhs) const {
        return p * rhs.q > rhs.p * q;
    }
    const rational operator+(const rational& rhs) const {
        return rational(p * rhs.q + q * rhs.p, q * rhs.q);
    }
    const rational operator-(const rational& rhs) const {
        return rational(p * rhs.q -q * rhs.p, q * rhs.q);
    }
    const rational operator*(const rational& rhs) const {
        return rational(p * rhs.p, q * rhs.q);
    }
    const rational operator/(const rational& rhs) const {
        return rational(p * rhs.q, q * rhs.p);
    }
};

inline rational FABS(rational r) {
    if (r < rational(0, 1)) {
        return r * -1;
    } else {
        return r;
    }
}

const double EPS = 1e-10;

typedef vector < int > VI;
typedef rational T ;
typedef vector < T > VT;
typedef vector < VT > VVT;

/*
   aX = b
   b is the column vector representing the solutions
   a is the coefficients of the linear equation
   X is just the column vector representing x1, x2, x3..... xN

   GaussJordan returns determinant and modifies b passed which stores the values of
   x1, x2...xN

   T is generics which is nice so just use whatever data type you want!!
Note: A must be square matrix!!
 */

T GaussJordan(VVT &a, VVT &b) {
    const int n = a.size();
    const int m = b[0].size();
    VI irow(n), icol(n), ipiv(n);
    T det = T(1, 1);

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++) if (!ipiv[j])
            for (int k = 0; k < n; k++) if (!ipiv[k])
                if (pj == -1 || FABS(a[j][k]) > FABS(a[pj][pk])) { pj = j; pk = k; }
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        if (pj != pk) det = det * -1;
        irow[i] = pj;
        icol[i] = pk;
        T c = T(a[pk][pk].q, a[pk][pk].p);
        det = det * a[pk][pk];
        a[pk][pk] = T(1, 1);
        for (int p = 0; p < n; p++) a[pk][p] = a[pk][p] * c;
        for (int p = 0; p < m; p++) b[pk][p] = b[pk][p] * c;
        for (int p = 0; p < n; p++) if (p != pk) {
            c = a[p][pk];
            a[p][pk] = 0;
            for (int q = 0; q < n; q++) a[p][q] = a[p][q] - (a[pk][q] * c);
            for (int q = 0; q < m; q++) b[p][q] = b[p][q] - (b[pk][q] * c);
        }
    }

    for (int p = n - 1; p >= 0; p--) if (irow[p] != icol[p]) {
        for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
    }

    return det;
}

map < string, int > id;
map < int, string > inv;

inline int transition(string res) {
    for (int i = 0; i < (int) res.size(); i++) {
        string cur = res.substr(i);
        if (id.count(cur)) {
            return id[cur];
        }
    }
    return 0;
}

inline void solve(string a, string b) {
    id.clear(); inv.clear();
    int n = (int) a.size();
    int index = 0;
    for (int i = 0; i <= n; i++) {
        string curPrefix = a.substr(0, i);
        if (id.count(curPrefix) == 0) {
            id[curPrefix] = index++;
            inv[index - 1] = curPrefix;
        }
    }
    for (int i = 0; i <= n; i++) {
        string curPrefix = b.substr(0, i);
        if (id.count(curPrefix) == 0) {
            id[curPrefix] = index++;
            inv[index - 1] = curPrefix;
        }
    }
    int m = (int)(id.size());
    VVT A(m, VT(m));
    VVT B(m, VT(1));
    for (int i = 0; i < m; i++) {
        B[i][0] = rational(0, 1);
    }
    B[id[a]][0] = rational(1, 1);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            A[i][j] = rational(0, 1);
        }
    }
    for (int i = 0; i < m; i++) {
        A[i][i] = rational(1, 1);
        if ((int) inv[i].size() == n) {
            continue;
        }
        A[i][transition(inv[i] + "H")] = A[i][transition(inv[i] + "H")] + rational(-1, 2);
        A[i][transition(inv[i] + "T")] = A[i][transition(inv[i] + "T")] + rational(-1, 2);
    }
    T det = GaussJordan(A, B);
    cout << B[0][0].p << '/' << B[0][0].q << '\n';
```

```
}


int main() {
    freopen("inp.in", "r", stdin);
    while (true) {
        string a; string b;
        cin >> a;
        if (a == "$") {
            return 0;
        }
        cin >> b;
        solve(a, b);
    }
}
```

6.7.3 Modular Gaussian Elimination

```
// SWERC 2016 - Problem I

#include "bits/stdc++.h"
using namespace std;
#define FOR(i,m,n) for(int i = (m); i < (n); i++)
#define ROF(i,m,n) for(int i = (n)-1; i >= (m); i--)
typedef long long LL;
typedef unsigned long long ULL;
typedef vector<int> VI;
typedef vector<LL> VLL;
#define SZ(x) ((int)(x).size())
#define MP make_pair
typedef pair<int,int> PII;
#define ll long long
typedef pair<LL,LL> PLL;
#define A first
#define B second
using namespace std;

inline long long gcd(long long a, long long b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

long long power(long long x, long long y, int mod=13)  {
    if (y == 1) {
        return x;
    }
    if (y == 0) {
        return 1;
    }
    long long res = power(x, y >> 1, mod);
    res = res * res;
    if (y & 1) {
        res *= x;
    }
    if (res >= mod) {
        res %= mod;
    }
    return res;
}
```

```
long long inv(long long x)  {
    return power(x, 11);
}

const double EPS = 1e-10;

typedef vector < int > VI;
typedef int T ;
typedef vector < T > VT;
typedef vector < VT > VVT;

/*
   aX = b
   b is the column vector representing the solutions
   a is the coefficients of the linear equation
   X is just the column vector representing x1, x2, x3..... xN
   GaussJordan returns determinant and modifies b passed which stores the values of
   x1, x2...xN
   T is generics which is nice so just use whatever data type you want

NOTE: a MUST BE a square Matrix

You can change the mod value in this function to anything you want, but it should be prime.
Appropriately, you should change the mod values in the power() and invert() functions above as well.
 */

void GaussJordan(VVT &a, VVT &b) {
    const int n = a.size();
    const int m = b[0].size();
    VI irow(n), icol(n), ipiv(n);

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++) if (!ipiv[j])
            for (int k = 0; k < n; k++) if (!ipiv[k])
                if (pj == -1 || abs(a[j][k]) > abs(a[pj][pk])) { pj = j; pk = k; }
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        irow[i] = pj;
        icol[i] = pk;
        T c = inv(a[pk][pk]);
        a[pk][pk] = 1;
        for (int p = 0; p < n; p++) {
            a[pk][p] = a[pk][p] * c;
            a[pk][p] %= 13;
        }
        for (int p = 0; p < m; p++) {
            b[pk][p] = b[pk][p] * c;
            b[pk][p] %= 13;
        }
        for (int p = 0; p < n; p++) if (p != pk) {
            c = a[p][pk];
            a[p][pk] = 0;
            for (int q = 0; q < n; q++) {
                a[p][q] = a[p][q] - (a[pk][q] * c) + (c * 13);
                a[p][q] %= 13;
            }
            for (int q = 0; q < m; q++) {
                b[p][q] = b[p][q] - (b[pk][q] * c) + (c * 13);
                b[p][q] %= 13;
            }
        }
    }

}

const int N = 505;
const int V = 205;

int n, a, r, t;
int edge_to_id[N][N];
PII id_to_edge[N];
int timer;
int adj[N][N];

int main() {
    scanf("%d %d %d %d", &n, &a, &r, &t);
    VVT A(t, VT(t));
    VVT B(t, VT(1));

    for (int i = 0; i < t; i++) {
        for (int j = 0; j < t; j++) {
            A[i][j] = 0;
        }
    }

    for (int j = 0; j < t; j++) {
        B[j][0] = 0;
    }

    memset(edge_to_id, -1, sizeof edge_to_id);

    for (int i = 0; i < t; i++) {
        int d, p; scanf("%d %d", &d, &p);
        B[i][0] = d;
        vector < int > cities;
        for (int j = 0; j < p; j++) {
            int city; scanf("%d", &city);
            cities.push_back(city);
        }
        for (int u = 0; u < p - 1; u++) {
            int v = u + 1;
            int p = cities[u];
            int q = cities[v];
            if (p > q) swap(p, q);
            if (edge_to_id[p][q] == -1) {
                edge_to_id[p][q] = timer++;
                id_to_edge[timer - 1] = MP(p, q);
            }
            A[i][edge_to_id[p][q]]++;
            A[i][edge_to_id[p][q]] %= 13;
        }
    }

    GaussJordan(A, B);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            adj[i][j] = 1e9;
        }
        adj[i][i] = 0;
    }
    for (int j = 0; j < t; j++) {
        int w = B[j][0];
        PII edge = id_to_edge[j];
        adj[edge.first][edge.second] = w;
        adj[edge.second][edge.first] = w;
    }
    for (int k = 1; k <= n; k ++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
            }
        }
    }
    printf("%d\n", adj[a][r]);
    return 0;
}
```

6.7.4 RREF

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting.  This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxm matrix
//
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPSILON = 1e-10;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

int rref(VVT &a) {
    int n = a.size();
    int m = a[0].size();
    int r = 0;
    for (int c = 0; c < m && r < n; c++) {
        int j = r;
        for (int i = r + 1; i < n; i++)
            if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
        if (fabs(a[j][c]) < EPSILON) continue;
        swap(a[j], a[r]);

        T s = 1.0 / a[r][c];
        for (int j = 0; j < m; j++) a[r][j] *= s;
        for (int i = 0; i < n; i++) if (i != r) {
            T t = a[i][c];
            for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
        }
        r++;
    }
    return r;
}

int main() {
    const int n = 5, m = 4;
    double A[n][m] = {
        {16,  2,  3, 13},
        { 5, 11, 10,  8},
        { 9,  7,  6, 12},
        { 4, 14, 15,  1},
        {13, 21, 21, 13}};
    VVT a(n);
    for (int i = 0; i < n; i++)
        a[i] = VT(A[i], A[i] + m);

    int rank = rref(a);

    // expected: 3
    cout << "Rank: " << rank << endl;

    // expected: 1 0 0 1
    //           0 1 0 3
    //           0 0 1 -3
    //           0 0 0 3.10862e-15
    //           0 0 0 2.22045e-15
    cout << "rref: " << endl;
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
}
```

6.7.5 XOR Shortest Path

```
/*
   Given some graph, find the shortest path from node 1, to node N (we can change
   these to anything)
   where the distance is the xor of the values of the edges on the path.

   This can be used to also solve the problem: given some numbers, determine
   if some subset of them is equal to some given value v in (log v) time
   also can find maximum value of any subset.
 */

#include <bits/stdc++.h>
using namespace std;
typedef long long int lli;

#define pb push_back
#define mp make_pair
#define MAX 100005
bool met[MAX];
int dist[MAX], bit[40];
vector<pair<int, int> > path[MAX];

void add_cycle(int v) {
    for (int i = 30; i >= 0; i--) {
        if (v & (1 << i)) {
            if (bit[i] == 0) {
                bit[i] = v;
                return;
            }
            v ^= bit[i];
        }
    }
}

void dfs(int node) {
    met[node] = true;
    for (pair<int, int> v : path[node]) {
        int adj = v.first;
        int w = v.second;
        if (met[adj]) {
            add_cycle(dist[node] ^ dist[adj] ^ w);
        } else {
            dist[adj] = dist[node] ^ w;
            dfs(adj);
        }
    }
}

int main() {

    int N, M; cin >> N >> M;
    for (int i = 0; i < M; i++) {
        int x, y, w; cin >> x >> y >> w;
        path[x].pb(mp(y, w));
        path[y].pb(mp(x, w));
    }

    dfs(1);
    int ans = dist[N];
    for (int i = 30; i >= 0; i--) {
        if (ans & (1 << i) && bit[i] != 0)
            ans ^= bit[i];
    }

    cout << ans << "\n";

    return 0;
}
```

6.8 Linear Programming

6.8.1 Simplex

```
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
```

```
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call solve(x).

#include "bits/stdc++.h"
using namespace std;

const long double EPS = 1e-9;

struct LPSolver {
    int m, n;
    vector < int > B, N;
    vector < vector < long double > > D;

    LPSolver(const vector < vector < long double > > &A, const vector < long double > &b, const vector < long double
    > &c) :
        m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, vector < long double >(n + 2)) {
            for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                    D[i][j] = A[i][j];
                }
            }
            for (int i = 0; i < m; i++) {
                B[i] = n + i;
                D[i][n] = -1;
                D[i][n + 1] = b[i];
            }
            for (int j = 0; j < n; j++) {
                N[j] = j;
                D[m][j] = -c[j];
            }
            N[n] = -1; D[m + 1][n] = 1;
        }

    inline void Pivot(int r, int s) {
        long double inv = 1.0 / D[r][s];
        for (int i = 0; i < m + 2; i++) {
            if (i != r) {
                for (int j = 0; j < n + 2; j++) {
                    if (j != s) {
                        D[i][j] -= D[r][j] * D[i][s] * inv;
                    }
                }
            }
        }
        for (int j = 0; j < n + 2; j++) {
            if (j != s) {
                D[r][j] *= inv;
            }
        }
        for (int i = 0; i < m + 2; i++) {
            if (i != r) {
                D[i][s] *= -inv;
            }
        }
        D[r][s] = inv;
        swap(B[r], N[s]);
    }

    inline bool Simplex(int phase) {
        int x = (phase == 1) ? (m + 1) : (m);
        while (true) {
            int s = -1;
            for (int j = 0; j <= n; j++) {
                if ((phase == 2) && (N[j] == -1)) {
                    continue;
                }
                if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) {
                    s = j;
                }
            }
            if (D[x][s] > -EPS) {
                return true;
            }
            int r = -1;
            for (int i = 0; i < m; i++) {
                if (D[i][s] < EPS) continue;
                if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
                        (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r = i;
            }
            if (r == -1) {
                return false;
            }
            Pivot(r, s);
        }
    }

    long double solve(vector < long double > &x) {
        int r = 0;
        for (int i = 1; i < m; i++) {
            if (D[i][n + 1] < D[r][n + 1]) {
                r = i;
            }
        }
        if (D[r][n + 1] < -EPS) {
            Pivot(r, n);
            if (!Simplex(1) || D[m + 1][n + 1] < -EPS) {
                return -numeric_limits < long double > :: infinity();
            }
            for (int i = 0; i < m; i++) {
                if (B[i] == -1) {
                    int s = -1;
                    for (int j = 0; j <= n; j++) {
                        if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
                    }
                    Pivot(i, s);
                }
            }
        }
        if (!Simplex(2)) {
            return numeric_limits<long double>::infinity();
        }
        x = vector < long double >(n);
        for (int i = 0; i < m; i++) {
            if (B[i] < n) {
                x[B[i]] = D[i][n + 1];
            }
        }
        return D[m][n + 1];
    }
};

const int N = 100;

int n;
int x[N], y[N];

inline int sq(int x) {
    return x * x;
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }
    vector < vector < long double > > a;
    vector < long double > b;
    vector < long double > c;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            vector < long double > constraint(n, 0.0);
            constraint[i] = constraint[j] = 1.0;
            a.push_back(constraint);
            long double allow = sqrt(sq(x[j] - x[i]) + sq(y[j] - y[i]));
            b.push_back(allow);
        }
        c.push_back(1);
    }
    LPSolver simplex(a, b, c);
    vector < long double > all_radius;
    simplex.solve(all_radius);
    long double sum_radii = 0.0;
    for (long double radius : all_radius) {
        sum_radii += radius;
    }
    cout << fixed << setprecision(10) << 2.0 * acos(-1) * sum_radii << "\n";
}
```

6.9 Matrix Exponentiation

6.9.1 Matrix Exponentiation Advanced

```
// IOITC 2015 Problem
/*
   DP[i] is a vector of length M denoting the number of arrays
   of length (i) for each 1 <= j <= M

   DP[2] = A * DP[1]
   DP[3] = B * DP[2] = B * (A * DP[1]) = (B * A) * DP[1]
   DP[4] = A * DP[3] = (A * (B * A) ) * DP[1]

   Let C = (B * A)

   A is an MxM matrix defined as follows
   for each (i, j) , if(j < i) A[i][j] = 1, else A[i][j] = 0

   B is an MxM matrix defines as follows
   for each (i, j) , if(j > i)  B[i][j] = 1, else B[i][j] = 0

   C is computed by multiplying B with A

   Ans = power(C, (N - 1) / 2)
   if( N is even ) Ans = multiply(A, Ans);
   We need to print sum of elements in Ans
   The complexity of this algorithm is O(M^3 log N)

   We need to find A[m]^K * C[m]^L * DP[1], where K <= 1, L <= 1000000 and m <= 40
   Precompute all powers of two of C[] matrix for each 1 <= m <= 40

   Now perform the multiplications in reverse order i.e. Instead of multiplying
   log N (MxM) matrices and multiplying the final result with a (Mx1) vector, start
   with multiplying the last (MxM) matrix with the (Mx1) vector and procees backwards.
   Thus you will keep getting a (Mx1) vector each time and you will perform only M^2
   operations while multiplying. Total complexity would abound to O(M ^ 2 log N)

 */

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

const int MAX = 41;
const int LN  = 21;
const int MOD = (int)(1e9 + 7);

/*-------- Matrix Data Type ---------*/
struct matrix{
    int mat[MAX][MAX];
    matrix(int m = MAX - 1){
        for(int i = 1; i <= m; i++)
            for(int j = 1; j <= m; j++)
                mat[i][j] = 0;
    }
};

matrix A[MAX], B[MAX], C[MAX][LN], unit, temp;
int Q, N, M;

/*---- Modulo Addition and Modulo Multiplication ----*/
inline int add(int x, int y){
    int res = x + y;
    if(res >= MOD) res -= MOD;
    return res;
}

inline int prod(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD) res %= MOD;
    return res;
}

/*--------- Matrix Multiplication of 2 Square Matrices ---------*/
inline matrix mult_sq(matrix p, matrix q, int m){
    matrix res(m);
    for(int i = 1; i <= m; i++)
        for(int k = 1; k <= m; k++)
            for(int j = 1; j <= m; j++)
                res.mat[i][j] = add(res.mat[i][j], prod(p.mat[i][k], q.mat[k][j]) );
    return res;
}

/*--------Precomputing Powers of C Matrix ---------*/
void precompute(int m){

    for(int i = 1; i <= m; i++)
        for(int j = 1; j < i; j++)
```

```
            A[m].mat[i][j] = 1;

    for(int i = 1; i <= m; i++)
        for(int j = i + 1; j <= m; j++)
            B[m].mat[i][j] = 1;

    C[m][0] = mult_sq(B[m], A[m], m);
    for(int i = 1; i < LN; i++)
        C[m][i] = mult_sq(C[m][i - 1], C[m][i - 1], m);
}


/*----- Multiplying a (m x m) and a (m x 1) matrix -----*/
matrix mult_sp(matrix x, matrix y, int m){
    matrix res(m);
    for(int i = 1; i <= m ; i++)
        for(int k = 1; k <= m ; k++)
            res.mat[i][1] = add(res.mat[i][1], prod(x.mat[i][k], y.mat[k][1]));
    return res;
}


// Note that this code might be rekt on local machine because of recursion with matrices
// It should work on judges tho

int main(){

    for(int i = 1; i <= 40; i++){
        unit.mat[i][i] = 1;
        precompute(i);
    }

    scanf("%d\n", &Q);

    while(Q--){

        scanf("%d %d\n", &N, &M);
        for(int i = 1 ; i <= M ; i++) temp.mat[i][1] = 1;

        int cur = (N - 1) >> 1;
        for(int i = 0 ; i <= 20 ; i++)
            if(cur & (1 << i)) temp = mult_sp(C[M][i], temp, M);

        if(N % 2 == 0) temp = mult_sp(A[M], temp, M);

        int res = 0;
        for(int i = 1; i <= M; i++) res = add(res, temp.mat[i][1]);
        printf("%d\n", res);
    }
}
```

6.9.2 Matrix Exponentiation

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

/*------ Matrix Exponentiation Template ------*/

const int ORD = 2; // Order of Square Matrix
const int MOD = 1000000007; // Modulo
```

```
inline int prod(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD) res %= MOD;
    return res;
}

inline int add(int x, int y){
    int res = x + y;
    if(res >= MOD) res -= MOD;
    return res;
}

struct matrix{
    int mat[ORD][ORD];
    matrix(){
        for(int i = 0; i < ORD; i++)
            for(int j = 0; j < ORD; j++)
                mat[i][j] = 0;
    }
    friend matrix operator * (matrix x, matrix y){
        matrix res;
        for(int i = 0; i < ORD; i++)
            for(int j = 0; j < ORD; j++)
                for(int k = 0; k < ORD; k++)
                    res.mat[i][j] = add(res.mat[i][j], prod(x.mat[i][k], y.mat[k][j]));
        return res;
    }
};

matrix base;

matrix exponentiate(matrix cur, long long p){
    matrix result;
    for (int i = 0; i < ORD; i++) {
        result.mat[i][i] = 1;
    }
    if (p == 0) {
        return result;
    }
    if (p == 1) {
        return base;
    }
    while (p > 0) {
        if (p & 1) {
            result = result * base;
        }
        base = base * base;
        p >>= 1;
    }
    return result;
}

// power(n) returns cur^{n}
matrix power(matrix cur, long long p) {
    base = cur;
    return exponentiate(cur, p);
}

/*------------ End of Template -------------*/

int main(){
    matrix fib;
    fib.mat[0][0] = 1, fib.mat[0][1] = 1;
    fib.mat[1][0] = 1, fib.mat[1][1] = 0;
    matrix result = power(fib, 10);
    cout << result.mat[0][0] << ' ' << result.mat[0][1] << endl;
    cout << result.mat[1][0] << ' ' << result.mat[1][1] << endl;
}
```

6.10 Miscellaneous Algebra

6.10.1 Algebra

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

#define MOD 1000000007
#define MAX 10000
typedef long long int lli;

// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a / b;
        int t = b; b = a%b; a = t;
        t = xx; xx = x - q*xx; x = t;
        t = yy; yy = y - q*yy; y = t;
    }
    return a;
}

// finds all solutions to ax = b (mod n)
vector<int> modular_linear_equation_solver(int a, int b, int n) {
    int x, y;
    vector<int> ret;
    int g = extended_euclid(a, n, x, y);
    if (!(b%g)) {
        x = mod(x*(b / g), n);
        for (int i = 0; i < g; i++)
            ret.push_back(mod(x + i*(n / g), n));
    }
    return ret;
}

// find z such that  z % m1 = r1, z % m2 = r2.
// Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M).  On failure, M = -1.
pair<int, int> chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
    int s, t;
    int g = extended_euclid(m1, m2, s, t);
    if (r1%g != r2%g) return make_pair(0, -1);
    return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
}

// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (m[i]).  Return (z, M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
```

```
pair<int, int> chinese_remainder_theorem(const vector<int> &m, const vector<int> &r) {
    pair<int, int> ret = make_pair(r[0], m[0]);
    for (int i = 1; i < m.size(); i++) {
        ret = chinese_remainder_theorem(ret.second, ret.first, m[i], r[i]);
        if (ret.second == -1) break;
    }
    return ret;
}

// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
    if (!a && !b) {
        if (c) return false;
        x = 0; y = 0;
        return true;
    }
    if (!a) {
        if (c % b) return false;
        x = 0; y = c / b;
        return true;
    }
    if (!b) {
        if (c % a) return false;
        x = c / a; y = 0;
        return true;
    }
    int g = gcd(a, b);
    if (c % g) return false;
    x = c / g * mod_inverse(a / g, b / g);
    y = (c - a*x) / b;
    return true;
}

// A O(n^2) time and O(n^2) extra space method for Pascal's Triangle
lli pascal_triangle[MAX][MAX];
void make_pascal(int n) {
    for (int line = 0; line < n; line++) {
        for (int i = 0; i <= line; i++) {
            if (line == i || i == 0)
                pascal_triangle[line][i] = 1;
            else
                pascal_triangle[line][i] = pascal_triangle[line-1][i-1] + pascal_triangle[line-1][i];
        }
    }
}

// create list of primitive pythagorean triples
vector<pair<pair<int, int>, int> > pyth;
void pythagorean_triples(int N) {
    for (int i = 1; i <= N; i++) {
        for (int k = 1; k < i; k++) {
            if (gcd(i,k) != 1) continue;
            if (i % 2 == 1 && k % 2 == 1) continue;

            int a = i*i - k*k;
            int b = 2*i*k;
            int c = i*i + k*k;
            pyth.push_back(make_pair(make_pair(a,b), c));
        }
    }
}
```

```
int eulerPhiDirect (int n) {
    int result = n;
    for (int i = 2; i <= n; i++) {
        if (is_prime[i] && result % i == 0) {
            result -= result / i;
        }
    }
    return result;
}

// euler totient. with sieve
int eulerPhi[1000];
void eulerSieve (int N) {
    for (int i = 1; i <= N; i++)
        eulerPhi[i] = i;
    for (int i = 1; i <= N; i++) {
        if (is_prime[i]) {
            for (int j = i; j <= N; j += i)
                eulerPhi[j] -= eulerPhi[j] / i;
        }
    }
}

// Fibonacci numbers.
/*      Useful formulas.

        Every positive integer can be represented uniquely as a sum of two or more
        distinct Fibonacci numbers. greedy.

        F(n+1) * F(n-1) - F(n) * F(n) = (-1) ^ n
        F(A+B) = F(A) * F(B+1) + F(A-1) * F(B)

        sum of first n fibonacci numbers = F(n+2) - 1

        gcd(F(n), F(m)) = F(gcd(n, m))
 */

// the n-th s-gonal number;
lli polygonal_number(int n, int s) {
    return ((s-2) * (n) * (n-1)) / 2 + n;
}

lli catalan_number(int n) {
    lli comb = pascal_comb_(2*n, n);
    return comb / (n+1);
}

// number of ways to partition a set of n objects into k non-empty subsets
lli secStirling[MAX][MAX];
lli second_kind_stirling (int n, int k) { // make sure to initialize secStirling to -1 for all values
    if (secStirling[n][k] >= 0) return secStirling[n][k];
    if (n == 0 && k == 0) return secStirling[n][k] = 1;
    else if (n == 0 || k == 0) return secStirling[n][k] = 0;
    return secStirling[n][k] = k * second_kind_stirling(n - 1, k) + second_kind_stirling(n - 1, k - 1);
}

lli firstStirling[MAX][MAX];
lli first_kind_stirling(int n, int k)  { // initialize firstStirling to -1 for all values
    if (firstStirling[n][k] >= 0) return firstStirling[n][k];
    if (n == 0 && k == 0) return firstStirling[n][k] = 1;
    else if (n == 0 || k == 0) return firstStirling[n][k] = 0;
    return firstStirling[n][k] = (n - 1) * first_kind_stirling(n - 1, k) + first_kind_stirling(n - 1, k - 1);
}

// evaluate (1^m + 2^m + ... + n^m) % MOD
int normal(int n) {
    n %= MOD;
    (n < 0) && (n += MOD);
    return n;
}

int add(int a, int b) { return a + b >= MOD ? a + b - MOD : a + b; }
int sub(int a, int b) { return a - b < 0 ? a - b + MOD : a - b; }
int mul(int a, int b) { return int(a * 1ll * b % MOD); }
int _div(int a, int b) { return modular_inverse_(a, b, MOD); }

int calc(const vector<int>& y, int x) {
    int ans = 0;
    int k = 1;
    for (int j = 1; j < (int)(y.size()); j++) {
        k = mul(k, normal(x - j));
        k = _div(k, normal(0 - j));
    }
    for (int i = 0; i < (int)(y.size()); i++) {
        ans = add(ans, mul(y[i], k));
        if (i + 1 >= y.size()) break;
        k = mul(k, _div(normal(x - i), normal(x - (i + 1))));
        k = mul(k, _div(normal(i - (y.size() - 1)), normal(i + 1)));
    }
    return ans;
}

int power_sum(int n, int k) {
    vector<int> y;
    int sum = 0;
    y.push_back(sum);
    for (int i = 0; i < k + 1; i++) {
        sum = add(sum, exponent_(i + 1, k, MOD));
        y.push_back(sum);
    }
    if (n < y.size()) return y[n];
    return calc(y, n);
}

/*
   Stirling's approximation:
   n! = sqrt(2 * PI * n) * (n/e) ^ n

   n-th harmonic number approximated by ln(n)
 */
```

6.10.2 Lagrange Multipliers

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


/*
   gradient of (function we want to minimize/maximize) = lambda * (gradient of (constraint function))
 */
```

```
/*
   Example problem:

   Suppose you are in a 2D world. Your are in a system
   of N parallel zones in which you are allowed
   to travel at different maximum speeds in any direction
   Width of each zone is 100 along the Y. You are currently at
   the origin and want to reach (100 * N, D)

   ----------------------------*goal
   max speed v3
   ----------------------------
   v2
   ----------------------------
   v1
   start* ----------------------------
 */

/*
Solution:
Say that you travel di meters in the x direction within zone i.
Then, we have the restriction that d1 + d2 + ... + dn = D
We want to minimize the sum of sqrt(di^2 + 100^2) / vi for i = 1 to n

We will need to binary search for the value of lambda
 */

const int MAX = 110;
int N, D;
double v[MAX];

double evaluate(double l) {
    double total = 0;
    for (int i = 0; i < N; i++) {
        double cur = sqrt((l*l *v[i]*v[i]* 100*100) / (1 - l*l*v[i]*v[i]));
        total += cur;
    }
    return total;
}

double get_time(double l) {
    double ans = 0;
    for (int i = 0; i < N; i++) {
        double dd = (l*l * v[i]*v[i]*100*100) / (1 - l*l*v[i]*v[i]);
        ans += sqrt(dd + 100*100) / v[i];
    }
    return ans;
}

int main() {

    int T; cin >> T;
    for (int t = 1; t <= T; t++) {
        cin >> N >> D;

        double mm = 0;
        for (int i = 0; i < N; i++) {
            cin >> v[i];
            mm = max(mm, v[i]);
        }
        double minim = 0, maxim = 1 / mm; // set correct bounds. Cant' have square root of a negative
        for (int aa = 0; aa < 400; aa++) {
            double lambda = (minim + maxim) / 2;
            double cur = evaluate(lambda);
            if (cur < D) {
                minim = lambda;
            } else {
                maxim = lambda;
            }
        }

        double lambda = minim;
        cout << fixed << setprecision(9);
        cout << "Case " << t << ": " << get_time(lambda) << "\n";
    }

    return 0;
}
```

6.11 Number Theory

6.11.1 Divisor Function

```
// Implementation of the sigma function for computing the number of divisors
// and the sum of the divisors of a number, given its prime factorization.

#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

ll NumberOfDivisors(const map<ll, ll>& factors) {
    ll ans = 1;
    for (auto pe : factors) ans *= pe.second + 1;
    return ans;
}

ll SumOfDivisors(const map<ll, ll>& factors) {
    ll ans = 1;
    for (auto pe : factors) {
        ll power_sum = 0;
        ll term = 1;
        for (ll i = 0; i <= pe.second; i++) {
            power_sum += term;
            term *= pe.first;
        }
        ans *= power_sum;
    }
    return ans;
}

int main() {
    map<ll, ll> factors = {{2, 10}, {3, 6}, {1009, 1}};
    assert(NumberOfDivisors(factors) == 154);
    assert(SumOfDivisors(factors) == 2259744710);
    return 0;
}
```

6.11.2 Euler Totient

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

/*
   Euler’s totient function phi(n) returns the number of positive integers less
   than or equal to n that are relatively prime to n. That is, phi(n) is the number
   of integers k in the range [1, n] for which gcd(n, k) = 1. The computation of
   phi(1..n) can be performed simultaneously, as done so by phi_table(n) which
   returns a vector v such that v[i] stores phi(i) for i in the range [0, n].

   Time Complexity:
   - O(n log(log(n))) per call to phi(n) and phi_table(n).

   Space Complexity:
   - O(1) auxiliary space for phi(n).
   - O(n) auxiliary heap space for phi_table(n).

   All credit goes to Alex Li.
 */

inline int phi(int n) {
    int res = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0) {
                n /= i;
            }
            res -= res / i;
        }
    }
    if (n > 1) {
        res -= res / n;
    }
    return res;
}

vector < int > phi_table(int n) {
    vector < int > res(n + 1);
    for (int i = 0; i <= n; i++) {
        res[i] = i;
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 2 * i; j <= n; j += i) {
            res[j] -= res[i];
        }
    }
    return res;
}

int main() {
    assert(phi(1) == 1);
    assert(phi(9) == 6);
    assert(phi(1234567) == 1224720);
    const int n = 1000;
    vector < int > v = phi_table(n);
    for (int i = 0; i <= n; i++) {
        assert(v[i] == phi(i));
    }
    return 0;
}
```

6.11.3 Extended Euclid

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

// extended_euclid(a, b) returns a pair (x, y) of integers such that gcd(a, b) = a * x + b * y.
inline pair < long long, long long > extended_euclid(long long a, long long b) {
    long long x = 1, y = 0, x1 = 0, y1 = 1;
    while (b != 0) {
        long long q = a / b, prev_x1 = x1, prev_y1 = y1, prev_b = b;
        x1 = x - q * x1;
        y1 = y - q * y1;
        b = a - q * b;
        x = prev_x1;
        y = prev_y1;
        a = prev_b;
    }
    return (a > 0) ? (make_pair(x, y)) : (make_pair(-x, -y));
}

int main() {
    pair < long long, long long > a = extended_euclid(30, 7);
    printf("%lld %lld\n", a.first, a.second);
}
```

6.11.4 Miller Rabin

```
// This is a deterministic variant of the Miller-Ragin primality test for
// signed 64-bit integers.

#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long ull;

const vector<int> p = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

inline ull MultiplyMod(ull x, ull n, ull m) {
    ull a = 0, b = x % m;
    for (; n > 0; n >>= 1) {
        if (n & 1) a = (a + b) % m;
        b = (b << 1) % m;
    }
    return a % m;
}

inline ull PowerMod(ull x, ull n, ull m) {
    ull a = 1, b = x;
    for (; n > 0; n >>= 1) {
        if (n & 1) a = MultiplyMod(a, b, m);
        b = MultiplyMod(b, b, m);
    }
    return a % m;
}

inline bool IsPrime(long long n) {
    for (int i = 0; i < p.size(); i++) {
        if (n % p[i] == 0) return n == p[i];
    }
    if (n < p.back()) return false;
    ull t = n - 1;
    int s = 0;
    for (t = n - 1; !(t & 1); t >>= 1) s++;
    for (int i = 0; i < p.size(); i++) {
        ull r = PowerMod(p[i], t, n);
        if (r == 1) continue;
        bool ok = false;
        for (int j = 0; j < s && !ok; j++) {
            ok |= (r == (ull)n - 1);
            r = MultiplyMod(r, r, n);
        }
        if (!ok) return false;
    }
    return true;
}

int main() {
    assert(IsPrime(1000000010000000029) == true);
    assert(IsPrime(5000000065000000021) == false);
    return 0;
}
```

6.12 Yates DP

6.12.1 Compatible Numbers

```
/*
   Compatible Numbers - Codeforces
   For each index i, find index j such that arr[i] & arr[j] == 0.
   Print -1 if such a j does not exist.

   N <= 1e6, 1 <= arr[i] <= 2**22

   Let f[i] be inverse of arr[i]. (Take only 22 bits!)
   Let ans[i] be the answer for number i.

   Notice that ans[f[i]] = arr[i].
   Moreover, you can switch off any subset of the on bits in f[i] and the answer
   would still be arr[i] for that resulting number. Naive method would take O(3 ^ N)
   but you can do it in O(2 ^ N * N) by iterating from MAX_VAL to 0, and switching off
   only one on bit at a time instead of all subsets.
 */

#include "bits/stdc++.h"
using namespace std;

const int LN = 22;
const int  N = 1 << LN;

int n;
int arr[N], dp[N];

inline int inv(int x){
    int res = x;
    for(int i = 0; i < LN; i++){
        res ^= (1 << i);
    }
    return res;
}
```

```
int main(){
    freopen("ioi.in", "r", stdin);
    fill(dp, dp + N, -1);
    scanf("%d", &n);
    for(int i = 1; i <= n; i++){
        scanf("%d", arr + i);
        dp[inv(arr[i])] = arr[i];
    }
    for(int i = N - 1; i >= 0; i--){
        if(dp[i] == -1) continue;
        for(int j = 0; j < LN; j++){
            if(i & (1 << j)){
                dp[i ^ (1 << j)] = dp[i];
            }
        }
    }
    for(int i = 1; i <= n; i++){
        printf("%d", dp[arr[i]]);
        if(i < n) printf(" ");
    }
}
```

6.12.2 Jzzhu and Numbers

```
/*
   Jzzhu and Numbers - Codeforces
   Number of subsets of an array of N elements whose bitwise-and is zero.
   N <= 1e6, arr[i] <= 1e6

   dp[mask][i] = Number of (x) in arr[] such that :-

   1) (arr[x] & mask) == (mask)
   2) arr[x] and mask might differ only the first i bits.

   The bits are numbered from right to left with the rightmost bit numbered (1).

   f[mask] = Number of (x) in arr[] such that (mask & arr[x] == mask)
   f[mask] = dp[mask][LN]

   f[0] = number of (x) in arr[] such that (arr[x] & 0 == 0) == n
   Answer = 2^n - 1
   but some of these subsets might have bitwise and with 1 bit set
   Answer -= 2^(f(x)) - 1 for all x such that popcount(x) == 1
   but some of these might have 2 bits set and we might have subtracted them twice...

   Thus, inclusion exclusion based on parity of popcount(x)!
 */

#include "bits/stdc++.h"
using namespace std;

const int LN = 20;
const int MOD = 1000000007;

int n;
int arr[1 << LN];
int dp[1 << LN][LN + 1];
int f[1 << LN], p[1 << LN];

int main(){
    freopen("ioi.in", "r", stdin);
    scanf("%d", &n);
    for(int i = 1; i <= n; i++){
        scanf("%d", arr + i);
        ++dp[arr[i]][0];
    }
    for(int mask = (1 << LN) - 1; mask >= 0; mask--){
        for(int i = 0; i < LN; i++){
            if(mask & (1 << i))
                dp[mask][i + 1] = dp[mask][i];
            else
                dp[mask][i + 1] = dp[mask][i] + dp[mask | (1 << i)][i];
        }
        f[mask] = dp[mask][LN];
    }
    p[0] = 1;
    for(int i = 1; i < (1 << LN); i++){
        p[i] = (2LL * p[i - 1]) % MOD;
    }
    for(int i = 0; i < (1 << LN); i++){
        p[i] = (p[i] - 1 + MOD) % MOD;
    }
    int ans = 0;
    for(int i = 0; i < (1 << LN); i++){
        if(__builtin_popcount(i) & 1)
            ans = (ans - p[f[i]] + MOD) % MOD;
        else
            ans = (ans + p[f[i]]) % MOD;
    }
    printf("%d\n", ans);
}
```

6.12.3 Special Pairs

```
/*
   Special Pairs - Hackerearth
   Find Number of unordered-pairs (i, j) such that A[i] & A[j] = 0
   Let a number be X. We know that for inverse of X, X will be a valid
   choice. Let X* be inverse of X, considering only the first LN bits.

   For all subsets of X*, X is a valid choice. We can consider all X**
   such that X** <= X*, and X will be a valid choice for each such X**.
   Hence, the problem reduces to finding # of X** in arr[]. This can be
   done with SOS-Subset DP in 2^(LN) * LN = N log N
 */

#include "bits/stdc++.h"
using namespace std;

const int LN = 20;
const int N  = (1 << LN);

int t, n;
int arr[N], freq[N], ans[N];
int dp[N][LN + 1];

int main(){
    freopen("ioi.in", "r", stdin);
    scanf("%d", &t);
    while(t--){
        memset(dp, 0, sizeof dp);
        memset(freq, 0, sizeof freq);
        memset(ans, 0, sizeof ans);
        scanf("%d", &n);
        for(int i = 1; i <= n; i++){
            scanf("%d", arr + i);
            freq[arr[i]]++;
        }
        for(int mask = 0; mask < N; mask++){
            dp[mask][0] = freq[mask];
            for(int i = 0; i < LN; i++){
                if(mask & (1 << i))
                    dp[mask][i + 1] = dp[mask][i] + dp[mask ^ (1 << i)][i];
                else
                    dp[mask][i + 1] = dp[mask][i];
            }
            ans[mask] = dp[mask][LN];
        }
        long long res = 0;
        for(int i = 1; i <= n; i++){
            res += ans[(1 << LN) - 1 - arr[i]]; // (2^(LN) - 1) ^ arr[i]
        }
        printf("%lld\n", res);
    }
}
```

7 NP Complete

7.1 Graph Coloring

```
/*
   Given an undirected graph, assign a color to every node such that no pair of
   adjacent nodes have the same color, and that the total number of colors used is
   minimized. color_graph() applies to a global, pre-populated adjacency matrix
   adj[][] which must satisfy the condition that adj[u][v] is true if and only if
   adj[v][u] is true, for all pairs of nodes u and v respectively between 0
   (inclusive) and the total number of nodes (exclusive) as passed in the function
   argument.

   Time Complexity:
   - Exponential on the number of nodes per call to color_graph().
   Space Complexity:
   - O(n^2) for storage of the graph, where n is the number of nodes.
   - O(n) auxiliary stack and heap space for color_graph().
 */

#include <algorithm>
#include <vector>

const int MAXN = 30;
int adj[MAXN][MAXN], min_colors, color[MAXN];
int curr[MAXN], id[MAXN + 1], degree[MAXN + 1];

void rec(int lo, int hi, int n, int used_colors) {
    if (used_colors >= min_colors) {
        return;
    }
    if (n == hi) {
        for (int i = lo; i < hi; i++) {
            color[id[i]] = curr[i];
        }
        min_colors = used_colors;
        return;
    }
    std::vector<bool> used(used_colors + 1);
    for (int i = 0; i < n; i++) {
        if (adj[id[n]][id[i]]) {
            used[curr[i]] = true;
        }
    }
    for (int i = 0; i <= used_colors; i++) {
        if (!used[i]) {
            int tmp = curr[n];
            curr[n] = i;
            rec(lo, hi, n + 1, std::max(used_colors, i + 1));
            curr[n] = tmp;
        }
    }
}

int color_graph(int nodes) {
    for (int i = 0; i <= nodes; i++) {
        id[i] = i;
        degree[i] = 0;
    }
    int res = 1, lo = 0;
    for (int hi = 1; hi <= nodes; hi++) {
        int best = hi;
        for (int i = hi; i < nodes; i++) {
            if (adj[id[hi - 1]][id[i]]) {
                degree[id[i]]++;
            }
            if (degree[id[best]] < degree[id[i]]) {
                best = i;
            }
        }
        std::swap(id[hi], id[best]);
        if (degree[id[hi]] == 0) {
            min_colors = nodes + 1;
            std::fill(curr, curr + nodes, 0);
            rec(lo, hi, lo, 0);
            lo = hi;
            res = std::max(res, min_colors);
        }
    }
    return res;
}

/*** Example Usage and Output:
  Colored using 3 color(s):
  Color 1: 0 3
  Color 2: 1 2
  Color 3: 4
 ***/

#include <cassert>
#include <iostream>
using namespace std;

void add_edge(int u, int v) {
    adj[u][v] = true;
    adj[v][u] = true;
}

int main() {
    add_edge(0, 1);
    add_edge(0, 4);
    add_edge(1, 3);
    add_edge(1, 4);
    add_edge(2, 3);
    add_edge(2, 4);
    add_edge(3, 4);
    int colors = color_graph(5);
    cout << "Colored using " << colors << " color(s):" << endl;
    for (int i = 0; i < colors; i++) {
        cout << "Color " << i + 1 << ":";
        for (int j = 0; j < 5; j++) {
            if (color[j] == i) {
                cout << " " << j;
            }
        }
        cout << endl;
    }
    return 0;
}
```

7.2 Max Clique

```
/*
   Given an undirected graph, max_clique() returns the size of the maximum clique,
   that is, the largest subset of nodes such that all pairs of nodes in the subset
   are connected by an edge. max_clique_weighted() additionally uses a global array
   w[] specifying a weight value for each node, returning the clique in the graph
   that has maximum total weight.
   Both functions apply to a global, pre-populated adjacency matrix adj[] which
   must satisfy the condition that adj[u][v] is true if and only if adj[v][u] is
   true, for all pairs of nodes u and v respectively between 0 (inclusive) and the
   total number of nodes (exclusive) as passed in the function argument. Note that
   max_clique_weighted() is an efficient implementation using bitmasks of unsigned
   64-bit integers, thus requiring the number of nodes to be less than 64.

   Time Complexity:
   - O(3^(n/3)) per call to max_clique() and max_clique_weighted(), where n
   is the number of nodes.
   Space Complexity:
   - O(n^2) for storage of the graph, where n is the number of nodes.
   - O(n) auxiliary stack space for max_clique() and max_clique_weighted().
 */

#include <algorithm>
#include <bitset>
#include <vector>

const int MAXN = 35;
typedef std::bitset<MAXN> bits;
typedef unsigned long long uint64;

bool adj[MAXN][MAXN];
int w[MAXN];

int rec(int nodes, bits &curr, bits &pool, bits &excl) {
    if (pool.none() && excl.none()) {
        return curr.count();
    }
    int ans = 0, u = 0;
    for (int v = 0; v < nodes; v++) {
        if (pool[v] || excl[v]) {
            u = v;
        }
    }
    for (int v = 0; v < nodes; v++) {
        if (!pool[v] || adj[u][v]) {
            continue;
        }
        bits ncurr, npool, nexcl;
        for (int i = 0; i < nodes; i++) {
            ncurr[i] = curr[i];
        }
        ncurr[v] = true;
        for (int j = 0; j < nodes; j++) {
            npool[j] = pool[j] && adj[v][j];
            nexcl[j] = excl[j] && adj[v][j];
        }
        ans = std::max(ans, rec(nodes, ncurr, npool, nexcl));
        pool[v] = false;
        excl[v] = true;
    }
    return ans;
}

int max_clique(int nodes) {
    bits curr, excl, pool;
    pool.flip();
    return rec(nodes, curr, pool, excl);
}

int rec(const std::vector<uint64> &g, uint64 curr, uint64 pool, uint64 excl) {
    if (pool == 0 && excl == 0) {
        int res = 0, u = __builtin_ctzll(curr);
        while (u < (int)g.size()) {
            res += w[u];
            u += __builtin_ctzll(curr >> (u + 1)) + 1;
        }
        return res;
    }
    if (pool == 0) {
        return -1;
    }
    int res = -1, pivot = __builtin_ctzll(pool | excl);
    uint64 z = pool & ~g[pivot];
    int u = __builtin_ctzll(z);
    while (u < (int)g.size()) {
        res = std::max(res, rec(g, curr | (1LL << u), pool & g[u], excl & g[u]));
        pool ^= 1LL << u;
        excl |= 1LL << u;
        u += __builtin_ctzll(z >> (u + 1)) + 1;
    }
    return res;
}

int max_clique_weighted(int nodes) {
    std::vector<uint64> g(nodes, 0);
    for (int i = 0; i < nodes; i++) {
        for (int j = 0; j < nodes; j++) {
            if (adj[i][j]) {
                g[i] |= 1LL << j;
            }
        }
    }
    return rec(g, 0, (1LL << nodes) - 1, 0);
}

/*** Example Usage ***/

#include <cassert>

void add_edge(int u, int v) {
    adj[u][v] = true;
    adj[v][u] = true;
}

int main() {
    add_edge(0, 1);
    add_edge(0, 2);
    add_edge(0, 3);
    add_edge(1, 2);
    add_edge(1, 3);
    add_edge(2, 3);
    add_edge(3, 4);
    add_edge(4, 2);
    w[0] = 10;
    w[1] = 20;
    w[2] = 30;
    w[3] = 40;
    w[4] = 50;
    assert(max_clique(5) == 4);
    assert(max_clique_weighted(5) == 120);
    return 0;
}
```

8 Strings

8.1 Aho Corasick

8.1.1 Aho Corasick

```
// SWERC 2016 - Passwords

#include "bits/stdc++.h"
using namespace std;

/*
   ------------ Aho-Corasick Template ----------

   This is an implementation of Aho-Corasick Algorithm to perform multiple pattern
   matching efficiently. The function aho_corasick() accepts a dictionary of strings
   and creates an automaton on those strings. Each node in the automaton represents
   a prefix of one of the strings in the dictionary. The root, indexed as 0, denotes
   the empty string.

   The following arrays are useful:
```

```
   - to[node][c]: Which node in the automaton would we be at if we append the character
   'c' to the end of the string represented by 'node'. This is the longest suffix
   of the new string that appears as a prefix in the automaton.

   - link[node]: Link to the longest proper suffix of the string represented by 'node'

   - isTerminal[node]: 'True' if any suffix (not necessarily proper) of 'node' exists
   in the dictionary provided to us, 'False' otherwise.

   Implementation Details:

   - M denotes the maximum number of nodes possible in the automaton, which is given by
   summation of lengths of all strings in our dictionary.

   - SIGMA denotes the size of our alphabet, which is usually 26.

   - INVALID denotes a transition character that does not appear in our alphabet. Use the
   variable "INVALID" in your code to denote such a character.

   Time Complexity:

   - O(length(s)) for add_str(s)
   - O(M * SIGMA) = O(M) for push_links()

   -----------------------------------------------
 */

const int M = 2000;
const int SIGMA = 26;
const int INVALID = SIGMA;

bool isTerminal[M];
int timer;
int to[M][SIGMA + 1], link[M];

inline void add_str(string s) {
    int cur = 0;
    for (char c: s) {
        if (!to[cur][c - 'a']) {
            to[cur][c - 'a'] = ++timer;
        }
        cur = to[cur][c - 'a'];
    }
    isTerminal[cur] = true;
}

void push_links() {
    queue < int > q;
    q.push(0);
    while(!q.empty()) {
        int V = q.front();
        q.pop();
        int U = link[V];
        if (!isTerminal[V]) {
            isTerminal[V] = isTerminal[U];
        }
        for (int c = 0; c < SIGMA; c++) {
            if (to[V][c]) {
                link[to[V][c]] = V ? to[U][c] : 0;
                q.push(to[V][c]);
            } else {
                to[V][c] = to[U][c];
            }
        }
    }
}

inline void aho_corasick(vector < string > patterns) {
    // Cleanup.
    timer = 0;
    memset(isTerminal, false, sizeof(isTerminal));
    memset(to, 0, sizeof(to));
    memset(link, 0, sizeof(link));
    // Add strings to automata.
    for (string s: patterns) {
        add_str(s);
    }
    for (int node = 0; node <= timer; node++) {
        // Appending an INVALID character from any node
        // should lead us back to the root node, giving us no match.
        to[node][INVALID] = 0;
    }
    // Compute Suffix Links.
    push_links();
}

// End of Aho Corasick Template.

const int N = 25;
const int MOD = 1000003;

inline int add(int x, int y) {
    int res = x + y;
    if (res >= MOD) {
        res -= MOD;
    }
    return res;
}

int dp[N][2][2][2][M];
map < char, int > leet;

int compute(int len, bool u, bool l, bool d, int node) {
    if (isTerminal[node]) {
        return 0;
    }
    int val = dp[len][u][l][d][node];
    if (val != -1) {
        return val;
    }
    if (len == 0) {
        return (u & l & d);
    }
    // Upper Case Letter
    val = 0;
    for (char c = 'A'; c <= 'Z'; c++) {
        int transition_node = to[node][c - 'A'];
        val = add(val, compute(len - 1, 1, l, d, transition_node));
    }
    // Lower Case Letter
    for (char c = 'a'; c <= 'z'; c++) {
        int transition_node = to[node][c - 'a'];
        val = add(val, compute(len - 1, u, 1, d, transition_node));
    }
    // Digit
    for (char c = '0'; c <= '9'; c++) {
        int transition_node = to[node][leet[c]];
        val = add(val, compute(len - 1, u, l, 1, transition_node));
    }
    return dp[len][u][l][d][node] = val;
}

inline int solve(int len) {
    int root = 0;
    return compute(len, 0, 0, 0, root);
}

int main() {
    ios :: sync_with_stdio(false);
    int l, r; cin >> l >> r;
    int n; cin >> n;
    vector < string > patterns;
    for (int i = 0; i < n; i++) {
        string x; cin >> x;
        patterns.push_back(x);
    }
    aho_corasick(patterns);
    for (char d = '0'; d <= '9'; d++) {
        leet[d] = INVALID;
    }
    leet['0'] = 'o' - 'a';
    leet['1'] = 'i' - 'a';
    leet['3'] = 'e' - 'a';
    leet['5'] = 's' - 'a';
    leet['7'] = 't' - 'a';
    memset(dp, -1, sizeof dp);
    int ans = 0;
    for (int i = l; i <= r; i++) {
        ans = add(ans, solve(i));
    }
    cout << ans << endl;
}
```

8.2 Hashing

8.2.1 2D Hashing

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

const int W = 2005;
const int MOD1 = 1000000007;
const int MOD2 = 1000000009;
const int BASE = 17;

int r, c;
int R, C;

char pat[W][W];
char arr[W][W];

pair < int, int > row[W][W];
pair < int, int > col[2][W];
pair < int, int > h[W * 10], p[W * 10];

// ------ Hashing Template -------

inline int prod1(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD1) res %= MOD1;
    return res;
}

inline int prod2(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD2) res %= MOD2;
    return res;
}

inline int add1(int x, int y){
    int res = x + y;
    if(res < 0) res += MOD1;
    if(res >= MOD1) res -= MOD1;
    return res;
}

inline int add2(int x, int y){
    int res = x + y;
    if(res < 0) res += MOD2;
    if(res >= MOD2) res -= MOD2;
    return res;
}

inline void build(string str){
    h[0] = {0, 0};
    int n = (int) str.size();
    for(int i = 1; i <= n; i++){
        h[i].first  = add1(prod1(h[i - 1].first, BASE), str[i - 1] - '0' + 1);
        h[i].second = add2(prod2(h[i - 1].second, BASE), str[i - 1] - '0' + 1);
    }
}

pair < int, int > getHash(int l, int r){
    pair < int, int > ans = {0, 0};
    ans.first  = add1(h[r].first,  -(prod1(h[l - 1].first, p[r - l + 1].first)));
    ans.second = add2(h[r].second, -(prod2(h[l - 1].second, p[r - l + 1].second)));
    return ans;
}

// ------ End of Hashing Template -------


// ------ KMP Template -------

const int MAX_LEN = 1e5 + 5;
int lps[MAX_LEN];

// lps[] table is 1 based, strings are 0 based.
inline void compute_table(vector < pair < int, int > > pattern) {
    lps[0] = -1, lps[1] = 0;
    int pref = 0;
    for (int i = 2; i <= pattern.size(); i++) {
        while (pref != -1 && pattern[i - 1] != pattern[pref]) {
            pref = lps[pref];
        }
        pref++;
        lps[i] = pref;
    }
}

// Function returns frequency of 'pattern' in 'text'
inline int kmp(vector < pair < int, int > > text, vector < pair < int, int > > pattern){
    compute_table(pattern);
    int pref = 0, count = 0;
    for (int i = 0; i < text.size(); i++) {
        while (pref != -1 && text[i] != pattern[pref]) {
            pref = lps[pref];
        }
        pref++;
        if (pref == pattern.size()) {
            pref = lps[pref];
            count++;
        }
    }
    return count;
}

// ---- End of KMP Template ----

int main(){
    ios :: sync_with_stdio(false);
    freopen ("inp.in", "r", stdin);

    cin >> r >> c >> R >> C;
    for (int i = 1; i <= r; i++) {
        for (int j = 1; j <= c; j++) {
            cin >> pat[i][j];
        }
    }
    for (int i = 1; i <= R; i++) {
        for (int j = 1; j <= C; j++) {
            cin >> arr[i][j];
        }
    }

    // Hash the rows
    p[0] = {1, 1};
    for(int i = 1; i < W * 10; i++){
        p[i].first  = prod1(p[i - 1].first, BASE);
        p[i].second = prod2(p[i - 1].second, BASE);
    }

    for (int i = 1; i <= R; i++) {
        string cur = "";
        for (int j = 1; j <= C; j++)
            cur += arr[i][j];
        build(cur);
        for (int j = 1; j <= C - c + 1; j++) {
            row[i][j] = getHash(j, j + c - 1);
        }
    }

    // Do KMP on columns
    vector < pair < int, int > > pattern;
    long long result = 0;

    for (int i = 1; i <= r; i++) {
        string cur = "";
        for (int j = 1; j <= c; j++)
            cur += pat[i][j];
        build(cur);
        pattern.push_back(getHash(1, c));
    }
    compute_table(pattern);

    for (int j = 1; j <= C - c + 1; j++) {
        vector < pair < int, int > > text;
        for (int i = 1; i <= R; i++) {
            text.push_back(row[i][j]);
        }
        result += kmp(text, pattern);
    }

    cout << result << endl;
}
```

8.2.2 String Hashing

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

const int MAX  = 5050;
const int MOD1 = 1000000007;
const int MOD2 = 1000000009;
const int BASE = 137;

int n, q, dp[MAX][MAX];
char str[MAX];
pair < int, int > h[MAX], rh[MAX], p[MAX];

inline int prod1(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD1) res %= MOD1;
    return res;
}

inline int prod2(int x, int y){
    long long res = x * 1LL * y;
    if(res >= MOD2) res %= MOD2;
    return res;
}

inline int add1(int x, int y){
    int res = x + y;
    if(res < 0) res += MOD1;
    if(res >= MOD1) res -= MOD1;
    return res;
}

inline int add2(int x, int y){
    int res = x + y;
    if(res < 0) res += MOD2;
    if(res >= MOD2) res -= MOD2;
    return res;
}
```

```
// Build tables
void build(){
    p[0] = {1, 1};
    for(int i = 1; i < MAX; i++){
        p[i].first  = prod1(p[i - 1].first, BASE);
        p[i].second = prod2(p[i - 1].second, BASE);
    }
    h[0] = {0, 0};
    for(int i = 1; i <= n; i++){
        h[i].first  = add1(prod1(h[i - 1].first, BASE), str[i] - 'a' + 1);
        h[i].second = add2(prod2(h[i - 1].second, BASE), str[i] - 'a' + 1);
    }
    rh[n + 1] = {0, 0};
    for(int i = n; i >= 1; i--){
        rh[i].first  = add1(prod1(rh[i + 1].first, BASE), str[i] - 'a' + 1);
        rh[i].second = add2(prod2(rh[i + 1].second, BASE), str[i] - 'a' + 1);
    }
}

// Returns hash of the substring [l, r]
pair < int, int > getHash(int l, int r){
    pair < int, int > ans = {0, 0};
    ans.first  = add1(h[r].first, -(prod1(h[l - 1].first, p[r - l + 1].first)));
    ans.second = add2(h[r].second,-(prod2(h[l - 1].second, p[r - l + 1].second)));
    return ans;
}

// Returns hash of the substring [r, l]
pair < int, int > getReverseHash(int l, int r){
    pair < int, int > ans = {0, 0};
    ans.first  = add1(rh[l].first, -(prod1(rh[r + 1].first,  p[r - l + 1].first)));
    ans.second = add2(rh[l].second,-(prod2(rh[r + 1].second, p[r - l + 1].second)));
    return ans;
}

bool isPalindrome(int i, int j){
    return getHash(i, j) == getReverseHash(i, j);
}

int main(){

    scanf("%s", str + 1);
    n = strlen(str + 1);
    scanf("%d", &q);

    build();

    dp[n][n] = 1;
    for(int i = n - 1; i >= 1; i--){
        dp[i][i] = 1;
        dp[i][i + 1] = 2 + (str[i] == str[i + 1]);
        for(int j = i + 2; j <= n; j++){
            dp[i][j] = dp[i + 1][j] + dp[i][j - 1] - dp[i + 1][j - 1];
            dp[i][j] += isPalindrome(i, j);
        }
    }

    while(q--){
        int xx, yy;
        scanf("%d %d", &xx, &yy);
        printf("%d\n", dp[xx][yy]);
    }
}
```

8.3 KMP

8.3.1 KMP

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


// ------ KMP Template -------

const int MAX_LEN = 1e5 + 5;
int lps[MAX_LEN];

// lps[] table is 1 based, strings are 0 based.
inline void compute_table(string &pattern) {
    lps[0] = -1, lps[1] = 0;
    int pref = 0;
    for (int i = 2; i <= pattern.size(); i++) {
        while (pref != -1 && pattern[i - 1] != pattern[pref]) {
            pref = lps[pref];
        }
        pref++;
        lps[i] = pref;
    }
}

// Function returns frequency of 'pattern' in 'text'
inline int kmp(string &text, string &pattern){
    compute_table(pattern);
    int pref = 0, count = 0;
    for (int i = 0; i < text.size(); i++) {
        while (pref != -1 && text[i] != pattern[pref]) {
            pref = lps[pref];
        }
        pref++;
        if (pref == pattern.size()) {
            pref = lps[pref];
            count++;
        }
    }
    return count;
}

// ---- End of KMP Template ----

int main() {
    string text, pattern;
    while (cin >> text >> pattern) {
        cout << kmp(text, pattern) << '\n';
    }
}
```

8.3.2 LCS KMP DP

```
/*
   Codeforces - Lucky Common Subsequence
   Prints Longest Common Subsequence of 's' and 't' that doesn't contain the substring 'pattern'
 */


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

const int N = 105;
const int L = 26;

string s, t, pattern;
int lps[N], f[N][L], dp[N][N][N];


/*
   f[i][j] = If I have matched the first "i" characters of "pattern", and I append
   character "j", what will be the new match length (lps) of the resulting string

   pattern => "ababa"
   f[3][b] = 4 --> This means I had an "aba" and I appended  a "b", now the new match length is "abab"
   f[3][a] = 1 --> This means I had an "aba" and I appended an "a", now the new match length is "a"

 */

inline void precompute() {
    for(int i = 1; i < pattern.size(); i++){
        int j = lps[i - 1];
        while (j > 0 and pattern[j] != pattern[i]){
            j = lps[j - 1];
        }
        j += pattern[i] == pattern[j];
        lps[i] = j;
    }
    for(int j = 0; j < 26; j++){
        f[0][j] = (pattern[0] - 'A') == j ? 1 : 0;
    }
    for(int i = 1; i < pattern.size(); i++){
        for(int j = 0; j < 26; j++){
            f[i][j] = (pattern[i] - 'A') == j ? i + 1 : f[lps[i - 1]][j];
        }
    }
    for (int j = 0; j < 26; j++) {
        f[pattern.size()][j] = f[lps[(int)pattern.size() - 1]][j];
    }
}

/*
   dp[i][j][match] = i and j denote positions in the string S1 and S2
   match denotes the length of the pattern which is the same as the suffix
   of our currently built string. It must never be equal to the entire pattern
   else we are violating our rule.
 */

inline int solve(int i, int j, int match){
    if(match == pattern.size()) return -(N * N);
    if(i == s.size() || j == t.size()) return 0;
    if(dp[i][j][match] != -1) return dp[i][j][match];
    int res = max(solve(i + 1, j, match), solve(i, j + 1, match));
    if(s[i] == t[j]){
        res = max(res, 1 + solve(i + 1, j + 1, f[match][s[i] - 'A']));
    }
    return dp[i][j][match] = res;
}

int main(){
    cin >> s >> t >> pattern;
    precompute();
    memset(dp, -1, sizeof dp);
    int lcs = solve(0, 0, 0);
    if(!lcs){
        cout << lcs << '\n';
        return 0;
    }
    int i = 0, j = 0, k = 0;
    while(lcs){
        int x = dp[i + 1][j][k];
        int y = dp[i][j + 1][k];
        int z = (s[i] == t[j]) ? (1 + dp[i + 1][j + 1][f[k][s[i] - 'A']]) : (0);
        if(x == dp[i][j][k]){
            i++;
        }
        else if(y == dp[i][j][k]){
            j++;
        }
        else{
            cout << s[i];
            k = f[k][s[i] - 'A'];
            lcs--, i++, j++;
        }
    }
}
```

8.3.3 Optimization KMP DP

```
/*

   Hackerearth - Benny and Two Strings
   Maximise occruences of pattern in text by modifying text following some constraints.
   KMP + DP problem

 */

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

const int N = 205;
const int K = 505;

string text, pattern;
int n, m, k;
int f[N][26], dp[N][N][K], lps[N];

inline int cost(int a, int b) {
    if (a > b) {
        swap(a, b);
    }
    int o1 = b - a;
    int o2 = a + 26 - b;
    return min(o1, o2);
}

/*
```

```
   dp[u][l][k] = The best I can do at position (u), current match length (l), cost remaining (k)
   l = x implies that in the currently built string, the last x characters are the same as the
   first x characters of "pattern".

 */

inline int solve(int u, int l, int k) {
    if (k < 0) {
        return -1e9;
    }
    if (u == text.size()) {
        return l == (int)pattern.size();
    }
    if (dp[u][l][k] != -1) {
        return dp[u][l][k];
    }
    int ans = -1e9;
    for (int i = 0; i < 26; ++i) {
        ans = max(ans, (l == (int)pattern.size()) + solve(u + 1, f[l][i], k - cost(text[u] - 'a', i)));
    }
    return dp[u][l][k] = ans;
}

/*
   f[i][j] = If I have matched the first "i" characters of "pattern", and I append
   character "j", what will be the new match length (lps) of the resulting string

   pattern => "ababa"
   f[3][b] = 4 --> This means I had an "aba" and I appended  a "b", now the new match length is "abab"
   f[3][a] = 1 --> This means I had an "aba" and I appended an "a", now the new match length is "a"

 */

inline void precompute() {
    for (int i = 1; i < pattern.size(); ++i) {
        int j = lps[i - 1];
        while (j > 0 and pattern[j] != pattern[i]) {
            j = lps[j - 1];
        }
        j += pattern[i] == pattern[j];
        lps[i] = j;
    }
    for (int j = 0; j < 26; ++j) {
        f[0][j] = (pattern[0] - 'a') == j ? 1 : 0;
    }
    for (int i = 1; i < pattern.size(); ++i) {
        for (int j = 0; j < 26; ++j) {
            f[i][j] = (pattern[i] - 'a') == j ? i + 1 : f[lps[i - 1]][j];
        }
    }
    for (int j = 0; j < 26; ++j) {
        f[pattern.size()][j] = f[lps[(int)pattern.size() - 1]][j];
    }
}

int main() {
    cin >> n >> m >> k;
    cin >> pattern >> text;
    precompute();
    memset(dp, -1, sizeof dp);
    cout << solve(0, 0, k) << '\n';
}
```

8.4 Suffix Arrays

8.4.1 Distinct Substrings

```
// SPOJ Distinct Substrings - len log^2 len

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

const int N  = 5e4 + 5;
const int LN = 18;

char str[N];

/*----------- Suffix Array Template ------------*/
// sa[i] = index of i'th smallest suffix in str[]
// "ana" --> {a, ana, na} -> sa[0] = 2, sa[1] = 0, sa[2] = 1

int pos[LN][N], sa[N], tmp[N];
int gap, len, level;

// Comparison function -> O(1)
inline bool suffix_cmp(int i, int j){
    if(pos[level][i] != pos[level][j]){
        return (pos[level][i] < pos[level][j]);
    }
    i += gap, j += gap;
    if(i < len && j < len){
        return (pos[level][i] < pos[level][j]);
    }
    return (i > j);
}

// Builds suffix array in len log^2 len
inline void build_suffix_array(){
    len = strlen(str);
    level = 0;
    for(int i = 0; i < len; i++){
        pos[level][i] = str[i];
        sa[i] = i;
    }
    for(gap = 1; ; gap *= 2){
        sort(sa, sa + len, suffix_cmp);
        for(int i = 1; i < len; i++){
            tmp[i] = tmp[i - 1] + suffix_cmp(sa[i - 1], sa[i]);
        }
        level = level + 1;
        for(int i = 0; i < len; i++){
            pos[level][sa[i]] = tmp[i];
        }
        if(tmp[len - 1] == len - 1) break;
    }
}

// Returns LCP of str[x..len-1] and str[y..len-1] in O(log len)
inline int lcp(int x, int y){
    int res = 0;
    for(int i = level; i >= 0; i--){
        if(x < len && y < len && pos[i][x] == pos[i][y]){
            res += (1 << i);
            x  += (1 << i);
            y  += (1 << i);
        }
    }
    return res;
}

/*-------------- End of Template --------------*/


inline void compute(){
    long long ans = len - sa[0];
    for(int i = 1; i < len; i++){
        ans += len - sa[i];
        ans -= lcp(sa[i - 1], sa[i]);
    }
    printf("%lld\n", ans);
}

int main(){
    int t;
    scanf("%d", &t);
    while(t--){
        scanf("%s", str);
        build_suffix_array();
        compute();
    }
}
```

8.4.2 Lexico Smallest Rotation

```
// ACM ICPC - Hidden Password

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

const int N  = 2e5 + 5;
const int LN = 20;

char str[N];

/*----------- Suffix Array Template ------------*/
// sa[i] = index of i'th smallest suffix in str[]
// "ana" --> {a, ana, na} -> sa[0] = 2, sa[1] = 0, sa[2] = 1

int pos[LN][N], sa[N], tmp[N];
int gap, len, level;

// Comparison function -> O(1)
inline bool suffix_cmp(int i, int j){
    if(pos[level][i] != pos[level][j]){
        return (pos[level][i] < pos[level][j]);
    }
    i += gap, j += gap;
    if(i < len && j < len){
        return (pos[level][i] < pos[level][j]);
    }
    return (i > j);
```

```
}

// Builds suffix array in len log^2 len
inline void build_suffix_array(){
    len = strlen(str);
    level = 0;
    for(int i = 0; i < len; i++){
        pos[level][i] = str[i];
        sa[i] = i;
    }
    for(gap = 1; ; gap *= 2){
        sort(sa, sa + len, suffix_cmp);
        for(int i = 1; i < len; i++){
            tmp[i] = tmp[i - 1] + suffix_cmp(sa[i - 1], sa[i]);
        }
        level = level + 1;
        for(int i = 0; i < len; i++){
            pos[level][sa[i]] = tmp[i];
        }
        if(tmp[len - 1] == len - 1) break;
    }
}

// Returns LCP of str[x..len-1] and str[y..len-1] in O(log len)
inline int lcp(int x, int y){
    int res = 0;
    for(int i = level; i >= 0; i--){
        if(x < len && y < len && pos[i][x] == pos[i][y]){
            res += (1 << i);
            x  += (1 << i);
            y  += (1 << i);
        }
    }
    return res;
}

/*----------- End of Template ------------*/


/* Returns the lexicographically smallest x length substring of str[]
   In case of multiple options, it chooses the one which has the lowest start_index */

inline void compute(int x){
    int st_idx = 0, idx = 0;
    for(int i = 0; i < len; i++){
        if(len - sa[i] >= x){
            st_idx = sa[i];
            idx = i;
            break;
        }
    }
    for(int i = idx + 1; i < len; i++){
        if(lcp(st_idx, sa[i]) >= x)
            st_idx = min(st_idx, sa[i]);
    }
    printf("%d\n", st_idx);
}

int main(){
    int t;
    scanf("%d", &t);
    while(t--){
        scanf("%d %s", &len, str);
        for(int i = len; i < len * 2; i++) str[i] = str[i - len];
        build_suffix_array();
        compute(len >> 1);
    }
}
```

8.5 Z Function

8.5.1 Substring Frequency ZFunction

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

const int N = 1e6 + 6;

char a[N], b[N], str[N * 2];
int t, z[N * 2];

inline void z_function(int n){
    memset(z, 0, sizeof z);
    for(int i = 1, l = 0, r = 0; i < n; i++){
        if(i <= r) z[i] = min(z[i - l], r - i + 1);
        while(i < n && str[z[i]] == str[i + z[i]]) ++z[i];
        if(i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
}

int main(){
    freopen("ioi.in", "r", stdin);
    scanf("%d", &t);
    for(int qq = 1; qq <= t; qq++){
        scanf("%s %s", a, b);
        int len1 = strlen(a), len2 = strlen(b);
        for(int i = 0; i < len2; i++) str[i] = b[i];
        str[len2] = '$';
        for(int i = 0; i < len1; i++) str[i + len2 + 1] = a[i];
        z_function(len1 + len2 + 1);
        int ans = 0;
        for(int i = len2 + 1; i < len2 + len1 + 1; i++) ans += (z[i] == len2);
        printf("Case %d: %d\n", qq, ans);
    }
}
```

8.5.2 Template ZFunction

```
// POI - Template

/*
   Print smallest length string which can be "stamped" multiple times to get
   Target String T

   Suppose T = ababbababbababbababbabababbaba
   Ans = 8, Ans_String = ababbaba

   The answer string will always be a prefix of T. Hence, you can compute z[i] for
   each index (i). Now you can do an offline algorithm to solve the problem.
 */
```

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

const int N = 5e5 + 50;
char str[N];
int n, z[N];
vector < pair < int, int > > values;

struct node{
    int mn, mx, ret;
    node(int _mn = -1, int _mx = -1, int _ret = -INT_MAX){
        mn  = _mn;
        mx  = _mx;
        ret = _ret;
    }
}tree[N * 4];

inline node merge(node x, node y){
    if(x.mn == -1 && x.mx == -1) return y;
    if(y.mn == -1 && y.mx == -1) return x;
    return node(x.mn, y.mx, max(x.ret, max(y.ret, y.mn - x.mx)));
}

inline void update(int i, int l, int r, int pos){
    if(l == r){
        tree[i].mn = tree[i].mx = l;
        return;
    }
    int mid = l + r >> 1;
    if(mid >= pos) update(i * 2, l, mid, pos);
    else update(i * 2 + 1, mid + 1, r, pos);
    tree[i] = merge(tree[i * 2], tree[i * 2 + 1]);
}

inline void z_function(){
    n = strlen(str);
    for(int i = 1, l = 0, r = 0; i < n; i++){
        if(i <= r) z[i] = min(z[i - l], r - i + 1);
        while(i < n && str[z[i]] == str[i + z[i]]) ++z[i];
        if(i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    z[0] = n;
}

int main(){
    scanf("%s", str);
    z_function();
    for(int i = 0; i < n; i++){
        if(z[i]) values.push_back(make_pair(z[i], i));
    }
    sort(values.begin(), values.end());
    reverse(values.begin(), values.end());
    for(int i = 0; i < n * 4; i++) tree[i] = node();
    int ans = n;
    update(1, 0, n + 1, 0), update(1, 0, n + 1, n + 1);
    for(int i = 0; i < (int) values.size(); i++){
        int cur = values[i].first, j = i + 1;
        while((j < (int) values.size()) && (values[j].first == cur)) j++;
        j--;
        for(int k = i; k <= j; k++){
            int idx = 1 + values[k].second;
            update(1, 0, n + 1, idx);
        }
        if(tree[1].ret <= cur) ans = cur;
        i = j;
    }
    printf("%d\n", ans);
}
```

9 Tool Box

9.1 BigIntegers

```
// Java BigInteger Implementation.

import java.math.*;
import java.util.*;
import java.io.*;

public class Main {

    public static void main(String[] args) {
        InputStream inputStream = System.in;
        OutputStream outputStream = System.out;
        InputReader in = new InputReader(inputStream);
        OutputWriter out = new OutputWriter(outputStream);
        Task solver = new Task();
        solver.solve(1, in, out);
        out.close();
    }

    static class Task {
        public void solve(int testNumber, InputReader in, OutputWriter out) {
            int n = in.readInt();
            for (int i = 0; i < n; i++) {
                BigInteger a = new BigInteger(in.readString());
                BigInteger b = new BigInteger(in.readString());
                BigInteger c = a.multiply(b);
                out.printLine(c.toString());
            }
        }
    }

    static class OutputWriter {
        private final PrintWriter writer;

        public OutputWriter(OutputStream outputStream) {
            writer = new PrintWriter(new BufferedWriter(new OutputStreamWriter(outputStream)));
        }

        public OutputWriter(Writer writer) {
            this.writer = new PrintWriter(writer);
        }

        public void print(Object... objects) {
            for (int i = 0; i < objects.length; i++) {
                if (i != 0) {
                    writer.print(' ');
                }
                writer.print(objects[i]);
            }
        }

        public void printLine(Object... objects) {
            print(objects);
            writer.println();
        }

        public void close() {
            writer.close();
        }
    }

    static class InputReader {
        private InputStream stream;
        private byte[] buf = new byte[1024];
        private int curChar;
        private int numChars;
        private InputReader.SpaceCharFilter filter;

        public InputReader(InputStream stream) {
            this.stream = stream;
        }

        public int read() {
            if (numChars == -1) {
                throw new InputMismatchException();
            }
            if (curChar >= numChars) {
                curChar = 0;
                try {
                    numChars = stream.read(buf);
                } catch (IOException e) {
                    throw new InputMismatchException();
                }
                if (numChars <= 0) {
                    return -1;
                }
            }
            return buf[curChar++];
        }

        public String readString() {
            int c = read();
            while (isSpaceChar(c)) {
                c = read();
            }
            StringBuilder res = new StringBuilder();
            do {
                if (Character.isValidCodePoint(c)) {
                    res.appendCodePoint(c);
                }
                c = read();
            } while (!isSpaceChar(c));
            return res.toString();
        }

        public int readInt() {
            int c = read();
            while (isSpaceChar(c)) {
                c = read();
            }
```

```
            int sgn = 1;
            if (c == '-') {
                sgn = -1;
                c = read();
            }
            int res = 0;
            do {
                if (c < '0' || c > '9') {
                    throw new InputMismatchException();
                }
                res *= 10;
                res += c - '0';
                c = read();
            } while (!isSpaceChar(c));
            return res * sgn;
        }


        public boolean isSpaceChar(int c) {
            if (filter != null) {
                return filter.isSpaceChar(c);
            }
            return isWhitespace(c);
        }

        public static boolean isWhitespace(int c) {
            return c == ' ' || c == '\n' || c == '\r' || c == '\t' || c == -1;
        }

        public String next() {
            return readString();
        }

        public interface SpaceCharFilter {
            public boolean isSpaceChar(int ch);
        }
    }
}
```

9.2 Date

```
// Routines for performing computations on dates.  In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (Julian day number)
int dateToInt (int m, int d, int y){
    return
        1461 * (y + 4800 + (m - 14) / 12) / 4 +
        367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
        3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
        d - 32075;
}

// converts integer (Julian day number) to Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
    int x, n, i, j;

    x = jd + 68569;
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay (int jd){
    return dayOfWeek[jd % 7];
}

int main (int argc, char **argv){
    int jd = dateToInt (3, 24, 2004);
    int m, d, y;
    intToDate (jd, m, d, y);
    string day = intToDay (jd);

    // expected output:
    //    2453089
    //    3/24/2004
    //    Wed
    cout << jd << endl
        << m << "/" << d << "/" << y << endl
        << day << endl;
}
```

9.3 Math Utilities

```
/*
   Base Conversion
   - Given an integer in base a as a vector d of digits (where d[0] is the least
   significant digit), convert_base(d, a, b) returns a vector of the integer's
   digits when converted base b (again with index 0 storing the least significant
   digit). The actual value of the entire integer to be converted must be able to
   fit within an unsigned 64-bit integer for intermediate storage.
   - convert_digits(x, b) returns the digits of the unsigned integer x in base b,
   where index 0 of the result stores the least significant digit.
   - to_roman(x) returns the Roman numeral representation of the unsigned integer x
   as a C++ string.
 */

std::vector<int> convert_base(const std::vector<int> &d, int a, int b) {
    unsigned long long x = 0, power = 1;
    for (int i = 0; i < (int)d.size(); i++) {
        x += d[i]*power;
        power *= a;
    }
    int n = ceil(log(x + 1)/log(b));
    std::vector<int> res;
    for (int i = 0; i < n; i++) {
        res.push_back(x % b);
        x /= b;
    }
    return res;
}

std::vector<int> convert_base(unsigned int x, int b = 10) {
    std::vector<int> res;
    while (x != 0) {
        res.push_back(x % b);
        x /= b;
    }
    return res;
}

std::string to_roman(unsigned int x) {
    static const std::string h[] =
    {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
    static const std::string t[] =
    {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
    static const std::string o[] =
    {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
    std::string prefix(x / 1000, 'M');
    x %= 1000;
    return prefix + h[x/100] + t[x/10 % 10] + o[x % 10];
}
```

9.4 String Parsing

```
// Splits a string 'work by the delimiters in the set 'delim', ignoring multiple occurences of each delimiter.
vector < string > split(string work, set < char > delim) {
    vector < string > flds;
    if (!flds.empty()) flds.clear();
    string buf = "";
    int i = 0;
    while (i < (int) work.size()) {
        bool is_delimiter = delim.count(work[i]);
        if (is_delimiter == false) {
            buf += work[i];
        } else if (((int) buf.size()) > 0) {
            flds.push_back(buf);
            buf = "";
        }
        i++;
    }
    if ((int) buf.size()) {
        flds.push_back(buf);
    }
    return flds;
}

// stod() -> converts a string to a double
// stoi() -> converts a string to a int
// stoll() -> converts a string to a long long

// Sometime, if we use getline() after using cin, we might require a cin.ignore() statement.
// Keep reading lines till EOF, stores string in s, reads SENTENCES, including SPACES.
while (getline(cin, s))
```