Team notebook

Md. Mottakin Chowdhury

June 16, 2018

Contents

1 DP

1.1 Convex Hull Trick

```
struct cht
{
	vector<pii> hull;
	vector<int> id;

	int cur=0;

	cht()
	{
		hull.clear();
		id.clear();
	}

	// Might need double here

	bool useless(const pii left, const pii middle, const pii right)
	{
		return
		1LL*(middle.second-left.second)*(middle.first-right.first)
		>=1LL*(right.second-middle.second)*(left.first-middle.first);
	}

	// Inserting line a*x+b with index idx
	// Before inserting one by one, all the lines are sorted by slope

	void insert(int idx, int a, int b)
	{
		if(hull.empty())
		{
			hull.pb(MP(a, b));
			id.pb(idx);
		}
		else
		{
			if(hull.back().first==a)
			{
				if(hull.back().second>=b)
				{
					return;
				}
				else
				{
					hull.pop_back();
					id.pop_back();
				}
			}
			while(hull.size()>=2 &&
			    useless(hull[hull.size()-2], hull.back(), MP(a,
```

```
                    b)))
                {
                    hull.pop_back();
                    id.pop_back();
                }
                hull.pb(MP(a,b));
                id.pb(idx);
            }
        }

        // returns maximum value and the index of the line
        // Pointer approach: the queries are sorted non-decreasing
        // Otherwise, we will need binary search

        pair<ll,int> query(int x)
        {
            ll ret=-INF;
            int idx=-1;
            for(int i=cur ; i < hull.size() ; i++)
            {
                ll tmp=1LL*hull[i].first*x + hull[i].second;

                if(tmp>ret)
                {
                    ret=tmp;
                    cur=i;
                    idx=id[i];
                }
                else
                {
                    break;
                }
            }
            return {ret,idx};
        }
};


// Slope decreasing, query minimum - Query point increasing.
// Slope increasing, query maximum - Query point increasing.
// Slope decreasing, query maximum - Query point decreasing.
// Slope increasing, query minimum - Query point decreasing.
```

1.2 Digit DP Sample 2

```
// For each case, output the case number and the number of integers in
    the range [A, B] which are
// divisible by K and the sum of its digits
// is also divisible by K.
int k, cases = 1;
ll dp[11][2][83][83];
int visited[11][2][83][83], flag;
string toString(int x)
{
        string temp = "";
        if (x == 0) return "0";
        while (x > 0)
        {
                int r = x % 10;
                temp = char(r + '0') + temp;
                x /= 10;
        }
        return temp;
}
ll calc(int idx, bool low, int modVal, int sumMod, string s)
{
        if (idx == s.size()) return (!modVal && !sumMod);
        if (visited[idx][low][modVal][sumMod] == flag)
                return dp[idx][low][modVal][sumMod];
        visited[idx][low][modVal][sumMod] = flag;
        int digit = low ? 9 : (s[idx] - '0');
        ll ret = 0;
        for (int i = 0; i <= digit; i++)
        {
                ret += calc(idx + 1, low || i < s[idx] - '0', (modVal * 10
                    + i) % k, (sumMod + i) % k, s);
        }
        return dp[idx][low][modVal][sumMod] = ret;
}
int main()
{
        int test;
        int a, b;
        cin >> test;
        while (test--)
        {
                cin >> a >> b >> k;
                if (k > 90)
```

```
        {
                cout << "Case " << cases++ << ": 0" << endl;
                continue;
        }
        string A = toString(a - 1);
        string B = toString(b);
        flag++;
        ll x = calc(0, 0, 0, 0, A);
        flag++;
        ll y = calc(0, 0, 0, 0, B);
        cout << "Case " << cases++ << ": " << y - x << endl;
    }
    return 0;
}
```

1.3 Digit DP Sample

```
// Calculate how many numbers in the range from A to B that have digit d
    in only the even positions and
// no digit occurs in the even position and the number is divisible by m.


string A, B; int m, d;
ll dp[2002][2002][2][2];

ll calc(int idx, int Mod, bool s, bool b)
{
        if(idx==B.size()) return Mod==0;

        if(dp[idx][Mod][s][b]!=-1)
                return dp[idx][Mod][s][b];

        ll ret=0;

        int low=s ? 0 : A[idx]-'0';
        int high=b ? 9 : B[idx]-'0';

        for(int i=low; i<=high; i++)
        {
                if(idx%2 && i!=d) continue;
                if(idx%2==0 && i==d) continue;
                ret=(ret+calc(idx+1, (Mod*10+i)%m, s || i>low, b ||
                    i<high))%mod;

                // if(ret>=mod) ret-=mod;
        }

        return dp[idx][Mod][s][b]=ret;
}

int main()
{
        // ios_base::sync_with_stdio(0);
        // cin.tie(NULL); cout.tie(NULL);
        // freopen("in.txt","r",stdin);

        cin>>m>>d>>A>>B;

        ms(dp,-1);

        prnt(calc(0,0,0,0));


        return 0;
}
```

1.4 Divide and Conquer DP

```
// http://codeforces.com/blog/entry/8219
// Divide and conquer optimization:
// Original Recurrence
//   dp[i][j] = min(dp[i-1][k] + C[k][j]) for k < j
// Sufficient condition:
//   A[i][j] <= A[i][j+1]
//   where A[i][j] = smallest k that gives optimal answer
// How to use:
//   // compute i-th row of dp from L to R. optL <= A[i][L] <= A[i][R] <=
    optR
//   compute(i, L, R, optL, optR)
//       1. special case L == R
//       2. let M = (L + R) / 2. Calculate dp[i][M] and opt[i][M] using
    O(optR - optL + 1)
//       3. compute(i, L, M-1, optL, opt[i][M])
//       4. compute(i, M+1, R, opt[i][M], optR)
```

```
// Example: http://codeforces.com/contest/321/problem/E
#include "../template.h"

const int MN = 4011;
const int inf = 1000111000;
int n, k;
ll cost[MN][MN], dp[811][MN];

inline ll getCost(int i, int j) {
    return cost[j][j] - cost[j][i-1] - cost[i-1][j] + cost[i-1][i-1];
}

void compute(int i, int L, int R, int optL, int optR) {
    if (L > R) return ;

    int mid = (L + R) >> 1, savek = optL;
    dp[i][mid] = inf;
    FOR(k,optL,min(mid-1, optR)) {
        ll cur = dp[i-1][k] + getCost(k+1, mid);
        if (cur < dp[i][mid]) {
            dp[i][mid] = cur;
            savek = k;
        }
    }
    compute(i, L, mid-1, optL, savek);
    compute(i, mid+1, R, savek, optR);
}

void solve() {
    cin >> n >> k;
    FOR(i,1,n) FOR(j,1,n) {
        cin >> cost[i][j];
        cost[i][j] = cost[i-1][j] + cost[i][j-1] - cost[i-1][j-1] +
            cost[i][j];
    }
    dp[0][0] = 0;
    FOR(i,1,n) dp[0][i] = inf;

    FOR(i,1,k) {
        compute(i, 1, n, 0, n);
    }
    cout << dp[k][n] / 2 << endl;
}
```

1.5 Dynamic Convex Hull Trick

```
// source:
    https://github.com/niklasb/contest-algos/blob/master/convex_hull/dynamic.cpp
// Used in problem CS Squared Ends

const ll is_query = -(1LL<<62);
struct Line {
    ll m, b;
    mutable function<const Line*()> succ;
    bool operator<(const Line& rhs) const {
        if (rhs.b != is_query) return m < rhs.m;
        const Line* s = succ();
        if (!s) return 0;
        ll x = rhs.m;
        return b - s->b < (s->m - m) * x;
    }
};
struct HullDynamic : public multiset<Line> { // will maintain upper hull
    for maximum
    bool bad(iterator y) {
        auto z = next(y);
        if (y == begin()) {
            if (z == end()) return 0;
            return y->m == z->m && y->b <= z->b;
        }
        auto x = prev(y);
        if (z == end()) return y->m == x->m && y->b <= x->b;

        // **** May need long double typecasting here
        return (long double)(x->b - y->b)*(z->m - y->m) >= (long
            double)(y->b - z->b)*(y->m - x->m);
    }
    void insert_line(ll m, ll b) {
        auto y = insert({ m, b });
        y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
        if (bad(y)) { erase(y); return; }
        while (next(y) != end() && bad(next(y))) erase(next(y));
        while (y != begin() && bad(prev(y))) erase(prev(y));
    }
    ll eval(ll x) {
        auto l = *lower_bound((Line) { x, is_query });
        return l.m * x + l.b;
    }
};
```

1.6 Edit Distance Recursive

```
int dp[34][34];
string a, b;

int editDistance(int i, int j)
{
	if (dp[i][j]!=-1)
		return dp[i][j];
	if (i==0)
		return dp[i][j]=j;
	if (j==0)
		return dp[i][j]=i;

	int cost;
	if (a[i-1]==b[j-1])
		cost=0;
	else
		cost=1;
	return
	    dp[i][j]=min(editDistance(i-1,j)+1,min(editDistance(i,j-1)+1,
	                                    editDistance(i-1,j-1)+cost));
}

int main()
{
	ms(dp,-1);
	cin>>a>>b;
	prnt(editDistance(a.size(),b.size()));
    return 0;
}
```

1.7 LCS

```
string a, b;
int dp[100][100];
string l;
void printLcs(int i, int j)
{
	if (a[i] == '\0' || b[j] == '\0')
	{
		cout << l << endl;
		return;
	}
	if (a[i] == b[j])
	{
		l += a[i];
		printLcs(i + 1, j + 1);
	}
	else
	{
		if (dp[i + 1][j] > dp[i][j + 1])
			printLcs(i + 1, j);
		else
			printLcs(i, j + 1);
	}
}
void printAll(int i, int j)
{
	if (a[i] == '\0' || b[j] == '\0')
	{
		prnt(l);
		return;
	}
	if (a[i] == b[j])
	{
		l += a[i];
		printAll(i + 1, j + 1);
		l.erase(l.end() - 1);
	}
	else
	{
		if (dp[i + 1][j] > dp[i][j + 1])
			printAll(i + 1, j);
		else if (dp[i + 1][j] < dp[i][j + 1])
			printAll(i, j + 1);
		else
		{
			printAll(i + 1, j);
			printAll(i, j + 1);
		}
	}
}
int lcslen (int i, int j)
```

```
{
        if (a[i] == '\0' || b[j] == '\0')
                return 0;
        if (dp[i][j] != -1)
                return dp[i][j];
        int ans = 0;
        if (a[i] == b[j])
        {
                ans = 1 + lcslen(i + 1, j + 1);
        }
        else
        {
                int x = lcslen(i, j + 1);
                int y = lcslen(i + 1, j);
                ans = max(x, y);
        }
        return dp[i][j] = ans;
}

int main()
{
        cin >> a >> b;
        ms(dp, -1);
        cout << lcslen(0, 0) << endl;
        printLcs(0, 0);
        l.clear();
        printAll(0, 0);
        return 0;
}
```

1.8 LIS nlogk

```
vector<int> d;
int ans, n;

int main() {
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int x;
        scanf("%d", &x);
        vector<int>::iterator it = lower_bound(d.begin(), d.end(), x);
        if (it == d.end()) d.push_back(x);
        else *it = x;
    }
    printf("LIS = %d", d.size());
    return 0;
}
```

1.9 Matrix Expo Class

```
struct Matrix
{
        ll mat[MAX][MAX];

        Matrix(){}

        // This initialization is important.
        // Input matrix should be initialized separately

        void init(int sz)
        {
                ms(mat,0);
                for(int i=0; i<sz; i++) mat[i][i]=1;
        }
} aux;

void matMult(Matrix &m, Matrix &m1, Matrix &m2, int sz)
{
        ms(m.mat,0);

        // This only works for square matrix

        FOR(i,0,sz)
        {
                FOR(j,0,sz)
                {
                        FOR(k,0,sz)
                        {
                                m.mat[i][k]=(m.mat[i][k]+m1.mat
                                        [i][j]*m2.mat[j][k])%mod;
                        }
                }
        }
}

Matrix expo(Matrix &M, int n, int sz)
```

```
{
	Matrix ret;
	ret.init(sz);

	if(n==0) return ret;
	if(n==1) return M;

	Matrix P=M;

	while(n!=0)
	{
		if(n&1)
		{
			aux=ret;
			matMult(ret,aux,P,sz);
		}

		n>>=1;

		aux=P; matMult(P,aux,aux,sz);
	}

	return ret;
}
```

1.10 Palindrome in a String

```
bool isPalindrome[100][100];
// Find the palindromes of a string in O(n^2)

int main()
{
	ios_base::sync_with_stdio(0);
	// freopen("in.txt","r",stdin);

	string s;

	cin>>s;

	int len=s.size();

	for(int i=0; i<len; i++)
		isPalindrome[i][i]=true;

	for(int k=1; k<len; k++)
	{
		for(int i=0; i+k<len; i++)
		{
			int j=i+k;

			isPalindrome[i][j]=(s[i]==s[j]) &&
			(isPalindrome[i+1][j-1] || i+1>=j-1);
		}
	}

	return 0;
}
```

2 Data Structures

2.1 Best Partial Sum in a Range

```
struct Node
{
	ll bestSum, bestPrefix, bestSuffix, segSum;
	Node()
	{
		bestSum=bestPrefix=bestSuffix=segSum=-INF;
	}
	void merge(Node &l, Node &r)
	{
		segSum=l.segSum+r.segSum;
		bestPrefix=max(l.bestPrefix,r.bestPrefix+l.segSum);
		bestSuffix=max(r.bestSuffix,r.segSum+l.bestSuffix);
		bestSum=max(max(l.bestSum,r.bestSum),l.bestSuffix+r.bestPrefix);
	}
}tree[150005];

void init(int node, int start, int end)
{
	if(start==end)
	{
		tree[node].bestSum=tree[node].segSum=a[start];
		tree[node].bestSuffix=tree[node].bestPrefix=a[start];
		return;
```

```
	}
	int left=node<<1;
	int right=left+1;
	int mid=(start+end)>>1;
	init(left,start,mid);
	init(right,mid+1,end);
	tree[node].merge(tree[left],tree[right]);
}
void update(int node, int start, int end, int i, int val)
{
	if(i<start || i>end)
		return;
	if(start>=i && end<=i)
	{
		tree[node].bestSum=tree[node].segSum=val;
		tree[node].bestSuffix=tree[node].bestPrefix=val;
		a[start]=val;
		return;
	}
	int left=node<<1;
	int right=left+1;
	int mid=(start+end)>>1;
	update(left,start,mid,i,val);
	update(right,mid+1,end,i,val);
	tree[node].merge(tree[left],tree[right]);
}
Node query(int node, int start, int end, int i, int j)
{
	if(i>end || j<start)
		return Node();
	if(start>=i && end<=j)
	{
		return tree[node];
	}
	int left=node<<1;
	int right=left+1;
	int mid=(start+end)>>1;
	Node l=query(left,start,mid,i,j);
	Node r=query(right,mid+1,end,i,j);
	Node n;
	n.merge(l,r);
	return n;
}
```

2.2 Binary Indexed Tree

```
ll Tree[MAX];

void update(int idx, ll x)
{
	while(idx<=n)
	{
		Tree[idx]+=x;
		idx+=(idx&-idx);
	}
}

ll query(int idx)
{
	ll sum=0;
	while(idx>0)
	{
		sum+=Tree[idx];
		idx-=(idx&-idx);
	}
	return sum;
}

int main()
{
	// For point update range query:
	// Point update: update(x,val);
	// Range query (a,b): query(b)-query(a-1);

	// For range update point query:
	// Range update (a,b): update(a,v); update(b+1,-v);
	// Point query: query(x);

	// Let's just consider only one update: Add v to [a, b] while the
	   rest elements of the array is 0.
	// Now, consider sum(0, x) for all possible x, again three
	   situation can arise:
	// 1. 0    x < a : which results in 0
	// 2. a    x    b : we get v * (x - (a-1))
	// 3. b < x < n : we get v * (b - (a-1))
	// This suggests that, if we can find v*x for any index x, then we
	   can get the sum(0, x) by subtracting T from it, where:
	// 1. 0    x < : Sum should be 0, thus, T = 0
	// 2. a    x    : Sum should be v*x-v*(a-1), thus, T = v*(a-1)
```

```
    // 3. b < x < n : Sum should be 0, thus, T = -v*b + v*(a-1)
    // As, we can see, knowing T solves our problem, we can use
        another BIT to store this additive amount from which we can
        get:
    // 0 for x < a, v*(a-1) for x in [a..b], -v*b+v(a-1) for x > b.

    // Now we have two BITs.
    // To add v in range [a, b]: Update(a, v), Update(b+1, -v) in the
        first BIT and Update(a, v*(a-1)) and Update(b+1, -v*b) on the
        second BIT.
    // To get sum in range [0, x]: you simply do Query_BIT1(x)*x -
        Query_BIT2(x);
    // Now you know how to find range sum for [a, b]. Just find sum(b)
        - sum(a-1) using the formula stated above.
  return 0;
}
```

2.3 Centroid Decomposition Sample

```
/* You are given a tree consisting of n vertices. A number is written on
    each vertex;
the number on vertex i is equal to a[i].
Let, g(x,y) is the gcd of the numbers written on the vertices belonging
    to the path from
x to y, inclusive. For i in 1 to 200000, count number of pairs (x,y)
    (1<=x<=y) such
that g(x,y) equals to i.
Note that 1<=x<=y does not really matter.
*/
vi graph[MAX];
int n, a[MAX], sub[MAX], total, cnt[MAX], cent, upto[MAX];
ll ans[MAX];
bool done[MAX];
set<int> take[MAX];

void dfs(int u, int p)
{
        sub[u]=1;
        total++;

        for(auto v: graph[u])
        {
                if(v==p || done[v]) continue;
                dfs(v,u);
                sub[u]+=sub[v];
        }
}

int getCentroid(int u, int p)
{
        // cout<<u<<" "<<sub[u]<<endl;
        for(auto v: graph[u])
        {
                if(!done[v] && v!=p && sub[v]>total/2)
                        return getCentroid(v,u);
        }

        return u;
}

void go(int u, int p, int val)
{
        ans[val]++;
        take[cent].insert(val);
        cnt[val]++;

        for(auto v: graph[u])
        {
                if(!done[v] && v!=p)
                {
                        go(v,u,upto[v]);
                }
        }
}

void calc(int u, int p, int val)
{
        for(auto it: take[cent])
        {
                int g=gcd(val,it);
                ans[g]+=cnt[it];
        }

        for(auto v: graph[u])
        {
                if(!done[v] && v!=p)
                {
                        calc(v,u,upto[v]);
```

```
            }
        }
}

void clean(int u, int p, int val)
{
        cnt[val]=0;

        for(auto v: graph[u])
        {
                if(!done[v] && v!=p)
                {
                        clean(v,u,upto[v]);
                }
        }
}

void calcgcd(int u, int p, int val)
{
        upto[u]=val;

        for(auto v: graph[u])
        {
                if(!done[v] && v!=p)
                {
                        calcgcd(v,u,gcd(val,a[v]));
                }
        }
}

void solve(int u)
{
        total=0;
        dfs(u,-1);

        cent=getCentroid(u,-1);
        calcgcd(cent,-1,a[cent]);

        // debug("cent",cent);
        done[cent]=true;

        for(auto v: graph[cent])
        {
                if(done[v]) continue;
                // cout<<"from centroid "<<cent<<" going to node:
                    "<<v<<endl;
                calc(v,cent,upto[v]);
                go(v,cent,upto[v]);
        }

        for(auto v: graph[cent])
        {
                if(!done[v])
                        clean(v,cent,upto[v]);
        }

        for(auto v: graph[cent])
        {
                if(!done[v])
                        solve(v);
        }
}

int main()
{
    // ios_base::sync_with_stdio(0);
    // cin.tie(NULL); cout.tie(NULL);
    // freopen("in.txt","r",stdin);

    int test, cases=1;

    scanf("%d", &n);
    FOR(i,1,n+1)
    {
        scanf("%d", &a[i]);
        ans[a[i]]++;
    }

    int u, v;

    FOR(i,1,n)
    {
        scanf("%d%d", &u, &v);
        graph[u].pb(v);
        graph[v].pb(u);
    }

    solve(1);
    FOR(i,1,MAX) if(ans[i]) printf("%d %lld\n", i, ans[i]);
```

```
    return 0;
}
```

2.4 Centroid Decomposition

```
int n, m, a, b, Table[MAX][20];
set<int> Graph[MAX];
int Level[MAX], nodeCnt, Subgraph[MAX], Parent[MAX], Ans[MAX];
void findLevel(int u)
{
        itrALL(Graph[u], it)
        {
                int v = *it;
                if (v != Table[u][0])
                {
                        Table[v][0] = u;
                        Level[v] = Level[u] + 1;
                        findLevel(v);
                }
        }
}
void Process()
{
        Level[0] = 0;
        ms(Table, -1);
        Table[0][0] = 0;
        findLevel(0);
// debug;
        for (int j = 1; 1 << j < n; j++)
        {
                for (int i = 0; i < n; i++)
                {
                        if (Table[i][j - 1] != -1)
                                Table[i][j] = Table[Table[i][j - 1]][j - 1];
                }
        }
// debug;
}
int findLCA(int p, int q)
{
        if (Level[p] < Level[q]) swap(p, q);
        int x = 1;
        while (true)
        {
                if ((1 << (x + 1)) > Level[p]) break;
                x++;
        }
        FORr(i, x, 0)
        {
                if (Level[p] - (1 << i) >= Level[q])
                        p = Table[p][i];
        }
        if (p == q) return p;
        FORr(i, x, 0)
        {
                if (Table[p][i] != -1 && Table[p][i] != Table[q][i])
                {
                        p = Table[p][i];
                        q = Table[q][i];
                }
        }
        return Table[p][0];
}
int Dist(int a, int b)
{
        return Level[a] + Level[b] - 2 * Level[findLCA(a, b)];
}
void findSubgraph(int u, int parent)
{
        Subgraph[u] = 1;
        nodeCnt++;
        itrALL(Graph[u], it)
        {
                int v = *it;
                if (v == parent) continue;
                findSubgraph(v, u);
                Subgraph[u] += Subgraph[v];
        }
}
int findCentroid(int u, int p)
{
        itrALL(Graph[u], it)
        {
                int v = *it;
                if (v == p) continue;
                if (Subgraph[v] > nodeCnt / 2) return findCentroid(v, u);
        }
```

```
        return u;
}
void Decompose(int u, int p)
{
        nodeCnt = 0;
        findSubgraph(u, u);
        int Cent = findCentroid(u, u);
        if (p == -1) p = Cent;
        Parent[Cent] = p;
        itrALL(Graph[Cent], it)
        {
                int v = *it;
                Graph[v].erase(Cent);
                Decompose(v, Cent);
        }
        Graph[Cent].clear();
}
void update(int u)
{
        int x = u;
        while (true)
        {
                Ans[x] = min(Ans[x], Dist(x, u));
                if (x == Parent[x]) break;
                x = Parent[x];
        }
}
int query(int u)
{
        int x = u;
        int ret = INF;
        while (true)
        {
                ret = min(ret, Dist(u, x) + Ans[x]);
                if (x == Parent[x]) break;
                x = Parent[x];
        }
        return ret;
}
int main()
{
// ios_base::sync_with_stdio(0);
// cin.tie(NULL); cout.tie(NULL);
// freopen("in.txt","r",stdin);
// All the nodes are initially blue
// Then by updating, one node is colored red
// Upon query, return the closest red node of the given node
        scanf("%d%d", &n, &m);
        FOR(i, 0, n - 1)
        {
                scanf("%d%d", &a, &b);
                a--, b--;
                Graph[a].insert(b);
                Graph[b].insert(a);
        }
        Process();
// debug;
        Decompose(0, -1);
        FOR(i, 0, n) Ans[i] = INF;
        update(0);
        while (m--)
        {
                int t, x;
                scanf("%d%d", &t, &x);
                x--;
                if (t == 1) update(x);
                else printf("%d\n", query(x));
        }
        return 0;
}
```

2.5 Counting Inversions with BIT

```
ll tree[200005];
int n, a[200005], b[200005];

void update(int idx, ll x)
{
        while(idx<=n)
        {
                tree[idx]+=x;
                idx+=(idx&-idx);
        }
}

int query(int idx)
{
        ll sum=0;
```

```
    while(idx>0)
    {
        sum+=tree[idx];
        idx-=(idx&-idx);
    }
    return sum;
}



int main()
{
    // ios_base::sync_with_stdio(0);
    // cin.tie(NULL); cout.tie(NULL); // No 'endl'
    // freopen("in.txt","r",stdin);
    int test;
    // cin>>test;
    scanf("%d", &test);
    while(test--)
    {
        ms(tree,0);
        scanf("%d", &n);
        FOR(i,1,n+1)
        {
            scanf("%d", &a[i]);
            b[i]=a[i];
        }

        sort(b+1,b+n+1);

        // Compressing the array
        FOR(i,1,n+1)
        {
            int rank=int(lower_bound(b+1,b+1+n,a[i])-b-1);
            a[i]=rank+1;
        }
        // FOR(i,1,n+1) cout<<a[i]<<" "; cout<<endl;
        ll ans=0;
        FORr(i,n,1)
        {
            ans+=query(a[i]-1);
            update(a[i],1);
        }
        // prnt(ans);
        printf("%lld\n",ans);
    }


    return 0;
}
```

2.6 How Many Values Less than a Given Value

```
// How many values in a range are less than or equal to the given value?
// The key idea is to sort the values under a node in the segment tree
    and use binary search to find
// the required count
// Complexity is O(nlog^2n) for building
// The actual problem needed the number of such values and the cumulative
    sum of them
// Tree[node].All has all the values and Tree[node].Pref has the prefix
    sums
// Remember: upper_bound gives the number of values less than or equal to
    given value in a sorted range
struct info
{
    vector<ll> All, Pref;
} Tree[MAX * 4];
ll T[MAX], Prefix[MAX];
void build(int node, int l, int r)
{
    if (l == r)
    {
        Tree[node].All.pb(T[l]);
        Tree[node].Pref.pb(T[l]);
        return;
    }
    int mid = (l + r) / 2;
    build(lc, l, mid);
    build(rc, mid + 1, r);
    for (auto it : Tree[lc].All)
        Tree[node].All.pb(it);
    for (auto it : Tree[rc].All)
        Tree[node].All.pb(it);
    SORT(Tree[node].All);
    ll now = 0;
    for (auto it : Tree[node].All)
    {
```

```
            Tree[node].Pref.pb(now + it);
            now += it;
        }
}
pair<ll, ll> query(int node, int l, int r, int x, int y, int val)
{
        if (x > r || y < l) return MP(0LL, 0LL);
        if (x <= l && r <= y)
        {
                int idx = upper_bound(Tree[node].All.begin(),
                    Tree[node].All.end(), val) - Tree[node].All.begin();
                if (idx > 0) return MP(Tree[node].Pref[idx - 1], idx);
                return MP(0LL, 0LL);
        }
        int mid = (l + r) / 2;
        pair<ll, ll> ret, left, right;
        left = query(lc, l, mid, x, y, val);
        right = query(rc, mid + 1, r, x, y, val);
        ret.first += left.first; ret.second += left.second;
        ret.first += right.first; ret.second += right.second;
        return ret;
}
```

2.7 Mo Algorithm Example

```
struct info
{
        int l, r, id;
        info(){}
        info(int l, int r, int id) : l(l), r(r), id(id){}
};

int n, t, a[2*MAX];
info Q[2*MAX];
int Block, cnt[1000004];
ll ans=0;
ll Ans[2*MAX];

inline bool comp(info a, info b)
{
        if(a.l/Block==b.l/Block) return a.r<b.r;
        return a.l<b.l;
}

inline void Add(int idx)
{
        ans+=(2*cnt[a[idx]]+1)*a[idx];
        cnt[a[idx]]++;

        /* Actual meaning of the above code

        ans-=cnt[a[idx]]*cnt[a[idx]]*a[idx];
        cnt[a[idx]]++;
        ans+=cnt[a[idx]]*cnt[a[idx]]*a[idx];

        */
}

inline void Remove(int idx)
{
        ans-=(2*cnt[a[idx]]-1)*a[idx];
        cnt[a[idx]]--;

        /* Actual meaning of the above code

        ans-=cnt[a[idx]]*cnt[a[idx]]*a[idx];
        cnt[a[idx]]--;
        ans+=cnt[a[idx]]*cnt[a[idx]]*a[idx];

        */
}

int main()
{
        // ios_base::sync_with_stdio(0);
        // cin.tie(NULL); cout.tie(NULL);
        // freopen("in.txt","r",stdin);

        // Problem: For each query, find the value cnt[a[i]]*cnt[a[i]]*a[i]

        scanf("%d%d", &n, &t);

        Block=sqrt(n);

        FOR(i,1,n+1) a[i]=getnum();

        FOR(i,0,t)
        {
```

```
        Q[i].l=getnum();
        Q[i].r=getnum();
        Q[i].id=i;
    }

    sort(Q,Q+t,comp);

    int Left=0, Right=-1;

    FOR(i,0,t)
    {
        while(Left<Q[i].l)
        {
            Remove(Left);
            Left++;
        }
        while(Left>Q[i].l)
        {
            Left--;
            Add(Left);
        }
        while(Right<Q[i].r)
        {
            Right++;
            Add(Right);
        }
        while(Right>Q[i].r)
        {
            Remove(Right);
            Right--;
        }

        Ans[Q[i].id]=ans;
    }

    FOR(i,0,t) printf("%lld\n", Ans[i]);

    return 0;
}
```

2.8 Mo on Tree Path

```
int aux[MAX], b[MAX], n, m, weight[MAX], u, v;
vi graph[MAX];
int parent[MAX][17], st[MAX], en[MAX], tag = 0, dist[MAX], blocSZ;
int go[100005], lca[100005], cnt[MAX], t[MAX];
bool seen[MAX];
struct info
{
    int u, v, id;
    bool fl;
    info() {}
    info(int u, int v, int id, bool fl) : u(u), v(v), id(id), fl(fl) {
        }
};
vector<info> Q;
// "Unordered"
void compress(int n, int *in, int *out)
{
    unordered_map <int, int> mp;
    for (int i = 1; i <= n; i++) out[i] = mp.emplace(in[i],
        mp.size()).first->second;
}
void dfs(int u, int p, int d)
{
    parent[u][0] = p;
    st[u] = ++tag;
    dist[u] = d;
    for (auto v : graph[u])
    {
        if (v != p) dfs(v, u, d + 1);
    }
    en[u] = ++tag;
    aux[st[u]] = u;
    aux[en[u]] = u;
}
void sparse()
{
    for (int j = 1; 1 << j < n; j++)
    {
        for (int i = 1; i <= n; i++)
        {
            if (parent[i][j - 1] != -1)
                parent[i][j] = parent[parent[i][j - 1]][j -
                    1];
        }
    }
}
```

```
int query(int p, int q)
{
	if (dist[p] < dist[q]) swap(p, q);
	int x = 1;
	while (true)
	{
		if ((1 << (x + 1)) > dist[p]) break;
		x++;
	}
	FORr(i, x, 0) if (dist[p] - (1 << i) >= dist[q]) p = parent[p][i];
	if (p == q) return p;
	FORr(i, x, 0)
	{
		if (parent[p][i] != -1 && parent[p][i] != parent[q][i])
		{
			p = parent[p][i];
			q = parent[q][i];
		}
	}
	return parent[p][0];
}
int ans = 0;
void doit(int idx)
{
	if (!seen[aux[idx]])
	{
		cnt[b[idx]]++;
		if (cnt[b[idx]] == 1) ans++;
	}
	else
	{
		cnt[b[idx]]--;
		if (cnt[b[idx]] == 0) ans--;
	}
	seen[aux[idx]] ^= 1;
}
int main()
{
	// Each node has some weight associated with it
	// u v : ask for how many different integers that represent the
	   weight of
	// nodes there are on the path from u to v.
	ms(parent, -1);
	scanf("%d%d", &n, &m);
	blocSZ = sqrt(n);
	FOR(i, 1, n + 1)
	{
		scanf("%d", &weight[i]);
	}
	FOR(i, 1, n)
	{
		scanf("%d%d", &u, &v);
		graph[u].pb(v);
		graph[v].pb(u);
	}
	dfs(1, 0, 0);
	sparse();
	compress(n, weight, t);
	(1, 1) << endl;
	FOR(i, 1, 2 * n + 1) b[i] = t[aux[i]];
	FOR(i, 0, m)
	{
		scanf("%d%d", &u, &v);
		lca[i] = query(u, v);
		if (st[u] > st[v]) swap(u, v);
		if (lca[i] == u) Q.pb(info(st[u], st[v], i, 0));
		else Q.pb(info(en[u], st[v], i, 1));
	}
	sort(Q.begin(), Q.end(), [](const info & a, const info & b)->bool
	{
		if (a.u / blocSZ == b.u / blocSZ) return a.v < b.v;
		return a.u < b.u;
	});
	int L = 1, R = 0;
	FOR(i, 0, Q.size())
	{
		int l = Q[i].u, r = Q[i].v, anc = lca[Q[i].id];

		while (R < r) { R++; doit(R); }
		while (R > r) { doit(R); R--; }
		while (L > l) { L--; doit(L); }
		while (L < l) { doit(L); L++; }

		if (Q[i].fl)
		{
			if (!cnt[b[st[anc]]])
				go[Q[i].id] = ans + 1;
			else go[Q[i].id] = ans;
		}
		else go[Q[i].id] = ans;
```

```
    }
    FOR(i, 0, m) printf("%d\n", go[i]);
    return 0;
}
```

2.9 Persistent Segment Tree 1

```
// Calculate how many distinct values are there in a given range
// Persistent Segment Tree implementation
// Actually used in Codeforces - The Bakery

int n, k, a[MAX], last[MAX], nxt[MAX];
int idx=1;
int Tree[64*MAX], L[64*MAX], R[64*MAX], root[2*MAX], rt[MAX];
int pos[MAX];

void build(int node, int l, int r)
{
    if(l==r)
    {
        Tree[node]=0;
        return;
    }

    L[node]=++idx;
    R[node]=++idx;

    // cout<<node<<" "<<L[node]<<" "<<R[node]<<endl;

    int mid=(l+r)/2;

    build(L[node],l,mid);
    build(R[node],mid+1,r);

    Tree[node]=0;
}

int update(int node, int l, int r, int pos, int val)
{
    int x;
    x=++idx;

    if(l==r)
    {
        Tree[x]=val;
        return x;
    }

    L[x]=L[node]; R[x]=R[node];

    int mid=(l+r)/2;

    if(pos<=mid) L[x]=update(L[x],l,mid,pos,val);
    else R[x]=update(R[x],mid+1,r,pos,val);

    Tree[x]=Tree[L[x]]+Tree[R[x]];

    return x;
}

int query(int node, int l, int r, int x, int y)
{
    if(x>r || y<l) return 0;
    if(x<=l && r<=y) return Tree[node];

    int mid=(l+r)/2;

    int q1=query(L[node],l,mid,x,y);
    int q2=query(R[node],mid+1,r,x,y);

    return q1+q2;
}

int getCost(int l, int mid)
{
    return query(root[rt[mid]],1,n,l,mid);
}

int main()
{
  int test, cases=1;

  scanf("%d%d", &n, &k);

  build(1,1,n);

  root[0]=1;
  int t=1;
```

```
    FOR(i,1,n+1)
    {
        scanf("%d", &a[i]);

        int k=pos[a[i]];

        if(!k)
        {
                root[t]=update(root[t-1],1,n,i,1);
                t++;
        }
        else
        {
                root[t]=update(root[t-1],1,n,k,0);
                t++;
                root[t]=update(root[t-1],1,n,i,1);
                t++;
        }

        rt[i]=t-1;
        pos[a[i]]=i;
    }

    return 0;
}
```

2.10 Persistent Segment Tree 2

```
const int MAXN = (1 << 20);

struct node
{
        int sum;
        node *l, *r;
        node() { l = nullptr; r = nullptr; sum = 0; }
        node(int x) { sum = x; l = nullptr; r = nullptr;   }
};

typedef node* pnode;

pnode merge(pnode l, pnode r)
{
        pnode ret = new node(0);
        ret->sum = l->sum + r->sum;
        ret->l = l;
        ret->r = r;
        return ret;
}

pnode init(int l, int r)
{
        if(l == r) { return (new node(0));}

        int mid = (l + r) >> 1;
        return merge(init(l, mid), init(mid + 1, r));
}

pnode update(int pos, int val, int l, int r, pnode nd)
{
        if(pos < l || pos > r) return nd;
        if(l == r) { return (new node(val)); }

        int mid = (l + r) >> 1;
        return merge(update(pos, val, l, mid, nd->l), update(pos, val, mid
            + 1, r, nd->r));
}

int query(int qL, int qR, int l, int r, pnode nd)
{
        if(qL <= l && r <= qR) return nd->sum;
        if(qL > r || qR < l) return 0;

        int mid = (l + r) >> 1;
    return query(qL, qR, l, mid, nd->l) + query(qL, qR, mid + 1, r,
        nd->r);
}

int get_kth(int k, int l, int r, pnode nd)
{
        if(l == r) return l;

        int mid = (l + r) >> 1;
        if(nd->l->sum < k) return get_kth(k - nd->l->sum, mid + 1, r,
            nd->r);
        else return get_kth(k, l, mid, nd->l);
}
```

2.11 Persistent Trie

```cpp
#include <bits/stdc++.h>

using namespace std;

// Problem: find maximum value (x^a[j]) in the range (l,r) where l<=j<=r

const int N = 1e5 + 100;
const int K = 15;

struct node_t;
typedef node_t * pnode;

struct node_t {
  int time;
  pnode to[2];
  node_t() : time(0) {
    to[0] = to[1] = 0;
  }
  bool go(int l) const {
    if (!this) return false;
    return time >= l;
  }
  pnode clone() {
    pnode cur = new node_t();
    if (this) {
      cur->time = time;
      cur->to[0] = to[0];
      cur->to[1] = to[1];
    }
    return cur;
  }
};

pnode last;
pnode version[N];

void insert(int a, int time) {
  pnode v = version[time] = last = last->clone();
  for (int i = K - 1; i >= 0; --i) {
    int bit = (a >> i) & 1;
    pnode &child = v->to[bit];
    child = child->clone();
    v = child;
    v->time = time;
  }
}

int query(pnode v, int x, int l) {
  int ans = 0;
  for (int i = K - 1; i >= 0; --i) {
    int bit = (x >> i) & 1;
    if (v->to[bit]->go(l)) { // checking if this bit was inserted before
        the range
      ans |= 1 << i;
      v = v->to[bit];
    } else {
      v = v->to[bit ^ 1];
    }
  }
  return ans;
}

void solve() {
  int n, q;
  scanf("%d %d", &n, &q);
  last = 0;
  for (int i = 0; i < n; ++i) {
    int a;
    scanf("%d", &a);
    insert(a, i);
  }
  while (q--) {
    int x, l, r;
    scanf("%d %d %d", &x, &l, &r);
    --l, --r;
    printf("%d\n", query(version[r], ~x, l));
    // Trie version[r] contains the trie for [0...r] elements
  }
}
```

2.12 Range Sum Query by Lazy Propagation

```cpp
int a[MAX + 7], tree[4 * MAX + 7], lazy[4 * MAX + 7];
void build(int node, int l, int r)
{
      if (l == r)
```

```
	{
		tree[node] = a[l];
		return;
	}
	if (l >= r) return;
	int mid = (l + r) / 2;
	build(node * 2, l, mid);
	build(node * 2 + 1, mid + 1, r);
	tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
void upd(int node, int l, int r, int v)
{
	lazy[node] += v;
	tree[node] += (r - l + 1) * x;
}
void pushDown(int node, int l, int r) //passing update information to the
    children
{
	int mid = (l + r) / 2;
	upd(node * 2, l, mid, lazy[node]);
	upd(node * 2 + 1, mid + 1, r, lazy[node]);
	lazy[node] = 0;
}
void update(int node, int l, int r, int x, int y, int v)
{
	if (x > r || y < l) return;
	if (x >= l && r <= y)
	{
		upd(node, l, r, v);
		return;
	}
	pushDown(node, l, r);
	int mid = (l + r) / 2;
	update(node * 2, l, mid, x, y, v);
	update(node * 2 + 1, mid + 1, r, x, y, v);
	tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

2.13 Splay Tree

```
/**
   Splay Tree :
   Node:
       void addIt(int ad) : adding an integer in a range
       void revIt() : reversing flag
       void upd() : push_up( gather from child)
       void pushdown() : pass values to the child( like lazy propagation)
   Splay:
       Node* newNode(int v,Node* f) :Returns Pointer of a node whose
           parent is f,and value v
       Node* build(int l,int r,Node* f) : building [L,R] which parent is f
       void rotate(Node* t,int d) : Rotation of Splay Tree
       void splay(Node* t,Node* f) : Splaying , t resides just below the f
       void select(int k,Node *f) : Select k th element in the tree
           ,splay it to the just below f
       Node*&get(int l, int r) : Getting The node for segment [L,R]
       void reverse(int l,int r) : Reverse a segment
       void del(int p) : deletes entry a[p]
       void split(int l,int r,Node*&s1) : Split the array and s1 stores
           the [L,R] segment
       void cut(int l,int r) : Cut the segment [L,R] and insert in at the
           end
       void insert(int p,int v): Insert after p,( 0 means before the
           array) an element whose value is v
       void insertRange(int pos,Node *s): Insert after pos, an segment
           denoted by s
       int query(int l,int r): Output desired result for [L,R]
       void addRange(int l,int r,int v): Add v to all the element in
           segment [L,R]
       void output(int l,int r) : Output the segment [L,R]
**/

/*
The following code answers the following queries
1 L R Output Maximum value in range [L,R]
2 L R Reverse the array [L,R]
3 L R v add v in range [L,R]
4 pos removes entry from pos
5 pos v - insert an element after position v

We assumes the initial array stored in ar[]={1,2,3,4... n}
*/

typedef int T;

const int N = 2e5+50; // >= Node + Query
T ar[N];          // Initial Array
struct Node{
```

```
    Node *ch[2],*pre; // child and parent
    T val; // Value stored in each node
    int size; //size of the subtree rooted at this node
    T mx; // additional info stored to solve problems, here maximum value
    T sum;
    T add;//lazy updates
    bool rev;// reverse flag
    Node(){size=0;val=mx=-1e9;add=0;}
    void addIt(T ad){
        add+=ad;
        mx+=ad;
        sum += size*ad;
        val+=ad;
    }
    void revIt(){
        rev^=1;
    }
    void upd(){
        size=ch[0]->size+ch[1]->size+1;
        mx=max(val,max(ch[0]->mx,ch[1]->mx));
        sum= ch[0]->sum + ch[1]->sum + val;
    }
    void pushdown();
}Tnull,*null=&Tnull;
void Node::pushdown(){
    if (add!=0){
        for (int i=0;i<2;++i)
            if (ch[i]!=null) ch[i]->addIt(add);
        add = 0;
    }
    if (rev){
        swap(ch[0],ch[1]);
        for (int i=0;i<2;i++)
            if (ch[i]!=null) ch[i]->revIt();
        rev = 0;
    }
}
struct Splay{
    Node nodePool[N],*cur; // Static Memory and cur pointer
    Node* root; // root of the splay tree
    Splay(){
        cur=nodePool;
        root=null;
    }
    void clear(){
        cur=nodePool;
        root=null;
    }
    Node* newNode(T v,Node* f){
        cur->ch[0]=cur->ch[1]=null;
        cur->size=1;
        cur->val=v;
        cur->mx=v;cur->sum = 0;
        cur->add=0;
        cur->rev=0;
        cur->pre=f;
        return cur++;
    }

    Node* build(int l,int r,Node* f){
        if(l>r) return null;
        int m=(l+r)>>1;
        Node* t=newNode(ar[m],f);
        t->ch[0]=build(l,m-1,t);
        t->ch[1]=build(m+1,r,t);
        t->upd();
        return t;
    }

    void rotate(Node* x,int c){
        Node* y=x->pre;
        y->pushdown();
        x->pushdown();

        y->ch[!c]=x->ch[c];
        if (x->ch[c]!=null) x->ch[c]->pre=y;
        x->pre=y->pre;
        if (y->pre!=null)
        {
            if (y->pre->ch[0]==y) y->pre->ch[0]=x;
            else y->pre->ch[1]=x;
        }
        x->ch[c]=y;
        y->pre=x;
        y->upd();
        if (y==root) root=x;
    }

    void splay(Node* x,Node* f){
```

```
    x->pushdown();
    while (x->pre!=f){
        if (x->pre->pre==f){
            if (x->pre->ch[0]==x) rotate(x,1);
            else rotate(x,0);
        }else{
            Node *y=x->pre,*z=y->pre;
            if (z->ch[0]==y){
                if (y->ch[0]==x) rotate(y,1),rotate(x,1);
                else rotate(x,0),rotate(x,1);
            }else{
                if (y->ch[1]==x) rotate(y,0),rotate(x,0);
                else rotate(x,1),rotate(x,0);
            }
        }
    }
    x->upd();
}
void select(int k,Node* f){
    int tmp;
    Node* x=root;
    x->pushdown();
    k++;
    for(;;){
        x->pushdown();
        tmp=x->ch[0]->size;
        if (k==tmp+1) break;
        if (k<=tmp) x=x->ch[0];
        else{
            k-=tmp+1;
            x=x->ch[1];
        }
    }
    splay(x,f);
}

Node*&get(int l, int r){
    select(l-1,null);
    select(r+1,root);
    return root->ch[1]->ch[0];
}

void reverse(int l,int r){
    Node* o=get(l,r);
    o->rev^=1;
    splay(o,null);
}
void del(int p)
{
        select(p-1,null);
        select(p+1,root);
        root->ch[1]->ch[0] = null;
        splay(root->ch[1],null);
}
void split(int l,int r,Node*&s1)
{
    Node* tmp=get(l,r);
    root->ch[1]->ch[0]=null;
    root->ch[1]->upd();
    root->upd();
    s1=tmp;
}
void cut(int l,int r)
{
    Node* tmp;
    split(l,r,tmp);
    select(root->size-2,null);
    root->ch[1]->ch[0]=tmp;
    tmp->pre=root->ch[1];
    root->ch[1]->upd();
    root->upd();
}

void init(int n){
    clear();
    root=newNode(0,null);
    root->ch[1]=newNode(n+1,root);
    root->ch[1]->ch[0]=build(1,n,root->ch[1]);
    splay(root->ch[1]->ch[0],null);
}


void insertPos(int pos,T v)
{
        select(pos,null);
        select(pos+1,root);
        root->ch[1]->ch[0] = newNode(v,root->ch[1]);
        splay(root->ch[1]->ch[0],null);
}
void insertRange(int pos,Node *s)
```

```
    {
        select(pos,null);
        select(pos+1,root);
        root->ch[1]->ch[0] = s;
        s->pre = root->ch[1];
        root->ch[1]->upd();
        root->upd();
    }
    T query(int l,int r)
    {
            Node *o = get(l,r);
            return o->mx;

    }
    void addRange(int l,int r,T v)
    {
            Node *o = get(l,r);
            o->add += v;
            o->val += v;
            o->sum += o->size * v;
            splay(o,null);

    }
    void output(int l,int r){
        for (int i=l;i<=r;i++){
            select(i,null);
            cout<<root->val<<endl;
        };
    }
}St;



int main()
{
    int n,m,a,b,c;

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

    for(int i= 1;i <= n;i ++ ) ar[i] = i;
    St.init(n);

    FOR(i,1,m+1)
    {
        scanf("%d%d", &a, &b);
        St.cut(a,b);
    }

    St.output(1,n);


    return 0;
}
```

3 Game

3.1 Green Hacenbush

```
// Green Hackenbush
vi graph[505];
int go(int u, int p)
{
        int ret = 0;
        for (auto &v : graph[u])
        {
                if (v == p) continue;
                ret ^= (go(v, u) + 1);
        }
        return ret;
}
int u, v, n;
int main()
{
// ios_base::sync_with_stdio(0);
// cin.tie(NULL); cout.tie(NULL);
// freopen("in.txt","r",stdin);
        int test, cases = 1;
        cin >> test;
        while (test--)
        {
                cin >> n;
                FOR(i, 0, n - 1)
                {
                        cin >> u >> v;
                        graph[u].pb(v);
                        graph[v].pb(u);
                }
```

```
            if (go(1, 0)) puts("Alice");
            else puts("Bob");
            FOR(i, 1, n + 1) graph[i].clear();
        }
        return 0;
}
```

3.2 Green Hackenbush 2

```
//
// Green Hackenbush
//
// Description:
//   Consider a two player game on a graph with a specified vertex (root).
//   In each turn, a player eliminates one edge.
//   Then, if a subgraph that is disconnected from the root, it is
     removed.
//   If a player cannot select an edge (i.e., the graph is singleton),
//   he will lose.
//
//   Compute the Grundy number of the given graph.
//
// Algorithm:
//   We use two principles:
//     1. Colon Principle: Grundy number of a tree is the xor of
//        Grundy number of child subtrees.
//        (Proof: easy).
//
//     2. Fusion Principle: Consider a pair of adjacent vertices u, v
//        that has another path (i.e., they are in a cycle). Then,
//        we can contract u and v without changing Grundy number.
//        (Proof: difficult)
//
//   We first decompose graph into two-edge connected components.
//   Then, by contracting each components by using Fusion Principle,
//   we obtain a tree (and many self loops) that has the same Grundy
//   number to the original graph. By using Colon Principle, we can
//   compute the Grundy number.
//
// Complexity:
//   O(m + n).
//
// Verified:
//   SPOJ 1477: Play with a Tree
//   IPSC 2003 G: Got Root?
//
//
#include <iostream>
#include <vector>
#include <cstdio>
#include <algorithm>
#include <functional>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())
#define TEST(s) if (!(s)) { cout << __LINE__ << " " << #s << endl;
    exit(-1); }

struct hackenbush {
        int n;
        vector<vector<int>> adj;

        hackenbush(int n) : n(n), adj(n) { }
        void add_edge(int u, int v) {
                adj[u].push_back(v);
                if (u != v) adj[v].push_back(u);
        }

        // r is the only root connecting to the ground
        int grundy(int r) {
                vector<int> num(n), low(n);
                int t = 0;
                function<int(int, int)> dfs = [&](int p, int u) {
                        num[u] = low[u] = ++t;
                        int ans = 0;
                        for (int v : adj[u]) {
                                if (v == p) { p += 2 * n; continue; }
                                if (num[v] == 0) {
                                        int res = dfs(u, v);
                                        low[u] = min(low[u], low[v]);
                                        if (low[v] > num[u]) ans ^= (1 + res)
                                            ^ 1; // bridge
                                        else                ans ^= res;
                                                    // non bridge
                                } else low[u] = min(low[u], num[v]);
```

```
                }
                if (p > n) p -= 2 * n;
                for (int v : adj[u])
                        if (v != p && num[u] <= num[v]) ans ^= 1;
                return ans;
            };
            return dfs(-1, r);
        }
};

int main() {
        int cases; scanf("%d", &cases);
        for (int icase = 0; icase < cases; ++icase) {
                int n; scanf("%d", &n);
                vector<int> ground(n);
                int r;
                for (int i = 0; i < n; ++i) {
                        scanf("%d", &ground[i]);
                        if (ground[i] == 1) r = i;
                }
                int ans = 0;
                hackenbush g(n);
                for (int i = 0; i < n - 1; ++i) {
                        int u, v;
                        scanf("%d %d", &u, &v);
                        --u; --v;
                        if (ground[u]) u = r;
                        if (ground[v]) v = r;
                        if (u == v) ans ^= 1;
                        else g.add_edge(u, v);
                }
                int res = ans ^ g.grundy(r);
                printf("%d\n", res != 0);
        }
}
```

4 Geometry

4.1 Convex Hull

```
struct PT
{
        int x, y;
        PT(){}
        PT(int x, int y) : x(x), y(y) {}
        bool operator < (const PT &P) const
        {
                return x<P.x || (x==P.x && y<P.y);
        }
};


ll cross(const PT p, const PT q, const PT r)
{
        return (ll)(q.x-p.x)*(ll)(r.y-p.y)-(ll)(q.y-p.y)*(ll)(r.x-p.x);
}

vector<PT> Points, Hull;

void findConvexHull()
{
        int n=Points.size(), k=0;

        SORT(Points);

        // Build lower hull

        FOR(i,0,n)
        {
                while(Hull.size()>=2 &&
                    cross(Hull[Hull.size()-2],Hull.back(),Points[i])<=0)
                {
                        Hull.pop_back();
                        k--;
                }
                Hull.pb(Points[i]);
                k++;
        }

        // Build upper hull

        for(int i=n-2, t=k+1; i>=0; i--)
        {
                while(Hull.size()>=t &&
                    cross(Hull[Hull.size()-2],Hull.back(),Points[i])<=0)
                {
```

```
                Hull.pop_back();
                k--;
            }
            Hull.pb(Points[i]);
            k++;
        }

        Hull.resize(k);
}
```

4.2 Counting Closest Pair of Points

```
int n;
struct Points
{
        double x, y;
        Points() {}
        Points(double x, double y) : x(x), y(y) { }
        bool operator<(const Points &a) const
        {
                return x < a.x;
        }
};
bool comp1(const Points &a, const Points &b)
{
        return a.x < b.x;
}
bool comp2(const Points &a, const Points &b)
{
        return a.y < b.y;
}
void printPoint(Points a)
{
        cout << a.x << " " << a.y << endl;
}
Points P[10005];
typedef set<Points, bool(*)(const Points&, const Points&)> setType;
typedef setType::iterator setIT;
setType s(&comp2);
double euclideanDistance(const Points &a, const Points &b)
{
// prnt((double)(a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
        return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
}
map<double, map<double, int> > CNT;
int main()
{
// ios_base::sync_with_stdio(0);
// cin.tie(NULL); cout.tie(NULL);
// freopen("in.txt","r",stdin);
        while ((cin >> n) && n)
        {
                FOR(i, 0, n) cin >> P[i].x >> P[i].y;
                sort(P, P + n, comp1);
                FOR(i, 0, n)
                {
// printPoint(P[i]);
                        s.insert(P[i]);
                        CNT[P[i].x][P[i].y]++;
                }
// To check repeated points :/
// for(auto it: s) printPoint(it);
                double ans = 10000;
                int idx = 0;
                FOR(j, 0, n)
                {
// cout<<"Point now: "; printPoint(P[j]);
                        if (CNT[P[j].x][P[j].y] > 1) ans = 0;
                        Points it = P[j];
                        while (it.x - P[idx].x > ans)
                        {
                                s.erase(P[idx]);
                                idx++;
                        }
                        Points low = Points(it.x, it.y - ans);
                        Points high = Points(it.x, it.y + ans);
                        setIT lowest = s.lower_bound(low);
                        if (lowest != s.end())
                        {
                                setIT highest = s.upper_bound(high);
                                for (setIT now = lowest; now != highest;
                                    now++)
                                {
                                        double cur = sqrt(euclideanDistance
                                                        (*now, it));
// prnt(cur);
                                        if (cur == 0) continue;
// cout<<"Here:"<<endl;
```

```
// printPoint(*now); printPoint(it); prnt
                                    (cur);
                                    if (cur < ans)
                                    {
                                           ans = cur;
                                    }
                             }
                      }
                      s.insert(it);
               }
// cout<<"Set now:"<<endl;
// for(auto I: s) printPoint(I);
               if (ans < 10000) cout << setprecision(4) << fixed << ans
                   << endl;
               else prnt("INFINITY");
               s.clear();
               CNT.clear();
        }
        return 0;
}
```

4.3 Maximum Points to Enclose in a Circle of Given Radius with Angular Sweep

```
typedef pair<double,bool> pdb;

#define START 0
#define END 1

struct PT
{
        double x, y;
        PT() {}
        PT(double x, double y) : x(x), y(y) {}
        PT(const PT &p) : x(p.x), y(p.y) {}
        PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
        PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
        PT operator * (double c)  const { return PT(x*c, y*c ); }
        PT operator / (double c)  const { return PT(x/c, y/c ); }
};

PT p[505];
double dist[505][505];
int n, m;

void calcDist()
{
        FOR(i,0,n)
        {
                FOR(j,i+1,n)
                        dist[i][j]=dist[j][i]=sqrt((p[i].x-p[j].x)*(p[i].x-p[j].x)
                                +(p[i].y-p[j].y)*(p[i].y-p[j].y));
        }
}

// Returns maximum number of points enclosed by a circle of radius
    'radius'
// where the circle is pivoted on point 'point'
// 'point' is on the circumfurence of the circle

int intelInside(int point, double radius)
{
        vector<pdb> ranges;

        FOR(j,0,n)
        {
                if(j==point || dist[j][point]>2*radius) continue;

                double a1=atan2(p[point].y-p[j].y,p[point].x-p[j].x);
                double a2=acos(dist[point][j]/(2*radius));

                ranges.pb({a1-a2,START});
                ranges.pb({a1+a2,END});
        }

        sort(ALL(ranges));

        int cnt=1, ret=cnt;

        for(auto it: ranges)
        {
                if(it.second) cnt--;
                else cnt++;
                ret=max(ret,cnt);
        }

        return ret;
}
```

```
// returns maximum amount of points enclosed by the circle of radius r
// Complexity: O(n^2*log(n))

int go(double r)
{
	int cnt=0;

	FOR(i,0,n)
	{
		cnt=max(cnt,intelInside(i,r));
	}

	return cnt;
}
```

4.4 Point in Polygon Binary Search

```
int sideOf(const PT &s, const PT &e, const PT &p)
{
	ll a = cross(e-s,p-s);
	return (a > 0) - (a < 0);
}

bool onSegment(const PT &s, const PT &e, const PT &p)
{
	PT ds = p-s, de = p-e;
	return cross(ds,de) == 0 && dot(ds,de) <= 0;
}

/*
Main routine
Description: Determine whether a point t lies inside a given polygon
    (counter-clockwise order).
The polygon must be such that every point on the circumference is visible
    from the first point in the vector.
It returns 0 for points outside, 1 for points on the circumference, and 2
    for points inside.
*/

int insideHull2(const vector<PT> &H, int L, int R, const PT &p) {
	int len = R - L;
	if (len == 2) {
		int sa = sideOf(H[0], H[L], p);
		int sb = sideOf(H[L], H[L+1], p);
		int sc = sideOf(H[L+1], H[0], p);
		if (sa < 0 || sb < 0 || sc < 0) return 0;
		if (sb==0 || (sa==0 && L == 1) || (sc == 0 && R ==
		    (int)H.size()))
		       return 1;
		return 2;
	}
	int mid = L + len / 2;
	if (sideOf(H[0], H[mid], p) >= 0)
		return insideHull2(H, mid, R, p);
	return insideHull2(H, L, mid+1, p);
}

int insideHull(const vector<PT> &hull, const PT &p) {
	if ((int)hull.size() < 3) return onSegment(hull[0], hull.back(),
	    p);
	else return insideHull2(hull, 1, (int)hull.size(), p);
}
```

4.5 Rectangle Union

```
struct info
{
	int x, ymin, ymax, type;
	info(){}
	info(int x, int ymin, int ymax, int type) :
		x(x), ymin(ymin), ymax(ymax), type(type) { }

	bool operator < (const info &p) const
	{
		return x<p.x;
	}
};

vector<info> in;
int n, x, y, p, q, m;
vi take;
int Lazy[4*MAX], Tree[4*MAX];

void update(int node, int l, int r, int ymin, int ymax, int val)
{
```

```
	if(take[l]>ymax || take[r]<ymin) return;

	if(ymin<=take[l] && take[r]<=ymax)
	{
		Lazy[node]+=val;

		if(Lazy[node]) Tree[node]=take[r]-take[l];
		else Tree[node]=Tree[lc]+Tree[rc];

		return;
	}

	if(l+1>=r) return;

	int mid=(l+r)/2;

	update(lc,l,mid,ymin,ymax,val);
	update(rc,mid,r,ymin,ymax,val);

	if(Lazy[node]) Tree[node]=take[r]-take[l];
	else Tree[node]=Tree[lc]+Tree[rc];
}

ll solve()
{
	take.clear(); ms(Tree,0); ms(Lazy,0);
	take.pb(-1);

	FOR(i,0,in.size())
	{
		take.pb(in[i].ymin);
		take.pb(in[i].ymax);
	}

	SORT(take);
	take.erase(unique(ALL(take)),take.end());
	m=take.size()-1;

	// VecPrnt(take);

	update(1,1,m,in[0].ymin,in[0].ymax,in[0].type);

	int prv=in[0].x; ll ret=0;

	FOR(i,1,in.size())
	{
		ret+=(ll)(in[i].x-prv)*Tree[1];
		prv=in[i].x;
		update(1,1,m,in[i].ymin,in[i].ymax,in[i].type);
	}

	return ret;
}

int main()
{
    // ios_base::sync_with_stdio(0);
    // cin.tie(NULL); cout.tie(NULL);
    // freopen("in.txt","r",stdin);

    int test, cases=1;

    scanf("%d", &test);

    while(test--)
    {
        scanf("%d", &n);

        in.clear();

        FOR(i,0,n)
        {
            scanf("%d%d%d%d", &x, &y, &p, &q);

            in.pb(info(x,y,q,1));
            in.pb(info(p,y,q,-1));
        }

        SORT(in);

        ll ans=solve();

        printf("Case %d: %lld\n", cases++, ans);
    }

    return 0;
}
```

4.6 Stanford ACM Team Geometry

```
double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y) {}
  PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)  const { return PT(x*c, y*c ); }
  PT operator / (double c)  const { return PT(x/c, y/c ); }
};

double dot(PT p, PT q)  { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  return os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p)  { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
```

```
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c,
       c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y -
           p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, 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;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion. Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
```

```
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

5 Graph

5.1 Articulation Points and Bridges

```
vi graph[100];
int dfs_num[100], dfs_low[100], parent[100], cnt;
int dfsroot, rootchild;
int art_v[100];

void articulate(int u)
{
	dfs_low[u]=dfs_num[u]=cnt++;
	for (ul j=0; j<graph[u].size(); j++)
	{
		int v=graph[u][j];
		if (dfs_num[v]==-1)
		{
			parent[v]=u;
			if (u==dfsroot)
				rootchild++;
			articulate(v);
			if (dfs_low[v]>=dfs_num[u])
				art_v[u]=true;
			if (dfs_low[v]>dfs_num[u])
				cout<<"Edge "<<u<<" & "<<v<<" is a
				     bridge."<<endl;
			dfs_low[u]=min(dfs_low[u],dfs_low[v]);
		}
		else if (v!=parent[u])
			dfs_low[u]=min(dfs_low[u],dfs_num[v]);
	}
}

int main()
{
	int n, m, u, v;
	cin>>n>>m;
	for (int i=0; i<m; i++)
	{
		cin>>u>>v;
		graph[u].pb(v);
		graph[v].pb(u);
	}
	cnt=0;
	ms(dfs_num,-1);
	for (int i=0; i<n; i++)
	{
		if (dfs_num[i]==-1)
		{
			dfsroot=i;
			rootchild=0;
			articulate(i);
			art_v[dfsroot]=(rootchild>1);
		}
	}
	prnt("Articulation points:");
	for (int i=0; i<n; i++)
	{
		if (art_v[i])
			cout<<"Vertex: "<<i<<endl;
	}
	return 0;
}
```

5.2 BCC

```
struct MagicComponents {

	struct edge {
		ll u, v, id;
	};

	ll num, n, edges;

	vector<ll> dfs_num, low, vis;
	vector<ll> cuts; // art-vertices
	vector<edge> bridges; // bridge-edges
	vector<vector<edge>> adj; // graph
	vector<vector<edge>> bccs; // all the bccs where bcc[i] has all
	    the edges inside it
	deque<edge> e_stack;

	// Nodes are numberd from 0

	MagicComponents(const ll& _n) : n(_n) {
		adj.assign(n, vector<edge>());
		edges = 0;
	}

	void add_edge(const ll& u, const ll& v) {
		adj[u].push_back({u,v,edges});
		adj[v].push_back({v,u,edges++});
	}

	void run(void) {
		vis.assign(n, 0);
		dfs_num.assign(n, 0);
		low.assign(n, 0);
		bridges.clear();
		cuts.clear();
		bccs.clear();
		e_stack = deque<edge>();
		num = 0;

		for (ll i = 0; i < n; ++i) {
			if (vis[i]) continue;
			dfs(i, -1);
		}
	}

	void dfs(const ll& node, const ll& par) {
		dfs_num[node] = low[node] = num++;
		vis[node] = 1;
		ll n_child = 0;
		for (edge& e : adj[node]) {
			if (e.v == par) continue;
			if (vis[e.v] == 0) {
				++n_child;
				e_stack.push_back(e);
				dfs(e.v, node);

				low[node] = min(low[node], low[e.v]);
				if (low[e.v] >= dfs_num[node]) {
					if (dfs_num[node] > 0 || n_child > 1)
						cuts.push_back(node);
					if (low[e.v] > dfs_num[node]) {
						bridges.push_back(e);

						pop(node);
					} else pop(node);
				}
			} else if (vis[e.v] == 1) {
				low[node] = min(low[node], dfs_num[e.v]);
				e_stack.push_back(e);
			}
		}
		vis[node] = 2;
	}

	void pop(const ll& u) {
		vector<edge> list;
		for (;;) {
			edge e = e_stack.back();
			e_stack.pop_back();
			list.push_back(e);
			if (e.u == u) break;
		}
		bccs.push_back(list);
	}

	//# Make sure to call run before calling this function.
	// Function returns a new graph such that all two connected
	// components are compressed into one node and all bridges
	// in the previous graph are the only edges connecting the
```

```
// components in the new tree.
// map is an integer array that will store the mapping
// for each node in the old graph into the new graph. //$
MagicComponents component_tree(vector<ll>& map) {
	vector<char> vis(edges);
	for (const edge& e : bridges)
		vis[e.id] = true;

	ll num_comp = 0;
	map.assign(map.size(), -1);
	for (ll i = 0; i < n; ++i) {
		if (map[i] == -1) {
			deque<ll> q;
			q.push_back(i);
			map[i] = num_comp;
			while (!q.empty()) {
				ll node = q.front();
				q.pop_front();
				for (const edge& e : adj[node]) {
					if (!vis[e.id] && map[e.v] ==
					    -1) {
						vis[e.id] = true;
						map[e.v] = num_comp;
						q.push_back(e.v);
					}
				}
			}
		}
		++num_comp;
	}

	MagicComponents g(num_comp);
	vis.assign(vis.size(), false);
	for (ll i = 0; i < n; ++i) {
		for (const edge& e : adj[i]) {
			if (!vis[e.id] && map[e.v] < map[e.u]) {
				vis[e.id] = true;
				g.add_edge(map[e.v], map[e.u]);
			}
		}
	}
	return g;
}

//# Make sure to call run before calling this function.
	// Function returns a new graph such that all biconnected
	// components are compressed into one node. Cut nodes will
	// be in multiple components, so these nodes will also have
	// their own component by themselves. Edges in the graph
	// represent components to articulation points
	// map is an integer array that will store the mapping
	// for each node in the old graph into the new graph.
	// Cut points to their special component, and every other node
	// to their specific component. //$
	MagicComponents bcc_tree(vector<ll>& map) {
		vector<ll> cut(n, -1);
		ll size = bccs.size();
		for (const auto& i : cuts)
			map[i] = cut[i] = size++;

		MagicComponents g(size);
		vector<ll> used(n);
		for (ll i = 0; i < bccs.size(); ++i) {
			for (const edge& e : bccs[i]) {
				vector<ll> tmp = {e.u,e.v};
				for (const ll& node : tmp) {
					if (used[node] != i+1) {
						used[node] = i+1;
						if (cut[node] != -1)
							g.add_edge(i,
							   cut[node]);
						else map[node] = i;
					}
				}
			}
		}
		return g;
	}
};
```

5.3 Bridges and Arts

```
struct MagicComponents {

	struct edge {
		ll u, v, id;
	};
```

```
ll num, n, edges;

vector<ll> dfs_num, low, vis;
vector<ll> cuts; // art-vertices
vector<edge> bridges; // bridges
vector<vector<edge>> adj; // graph
vector<vector<edge>> bccs; // contains the bccs, each bccs[i]
    contiains all the edges in a bcc
deque<edge> e_stack;

MagicComponents(const ll& _n) : n(_n) {
	adj.assign(n, vector<edge>());
	edges = 0;
}

void add_edge(const ll& u, const ll& v) {
	adj[u].push_back({u,v,edges});
	adj[v].push_back({v,u,edges++});
}

void run(void) {
	vis.assign(n, 0);
	dfs_num.assign(n, 0);
	low.assign(n, 0);
	bridges.clear();
	cuts.clear();
	bccs.clear();
	e_stack = deque<edge>();
	num = 0;

	for (ll i = 0; i < n; ++i) {
		if (vis[i]) continue;
		dfs(i, -1);
	}
}

void dfs(const ll& node, const ll& par) {
	dfs_num[node] = low[node] = num++;
	vis[node] = 1;
	ll n_child = 0;
	for (edge& e : adj[node]) {
		if (e.v == par) continue;
		if (vis[e.v] == 0) {
			++n_child;
			e_stack.push_back(e);
			dfs(e.v, node);

			low[node] = min(low[node], low[e.v]);
			if (low[e.v] >= dfs_num[node]) {
				if (dfs_num[node] > 0 || n_child > 1)
					cuts.push_back(node);
				if (low[e.v] > dfs_num[node]) {
					bridges.push_back(e);
					pop(node);
				} else pop(node);
			}
		} else if (vis[e.v] == 1) {
			low[node] = min(low[node], dfs_num[e.v]);
			e_stack.push_back(e);
		}
	}
	vis[node] = 2;
}

void pop(const ll& u) {
	vector<edge> list;
	for (;;) {
		edge e = e_stack.back();
		e_stack.pop_back();
		list.push_back(e);
		if (e.u == u) break;
	}
	bccs.push_back(list);
}
};
```

5.4 Dijkstra!

```
struct road
{
	int u, w;
	road (int a, int b)
	{
		u=a; w=b;
	}
	bool operator < (const road & p) const
	{
		return w>p.w;
```

```
        }
};

int d[100], parent[100], start, end;
mvii g, cost;

void dijkstra (int n)
{
        ms(d,INF);
        ms(parent,-1);
        priority_queue <road> Q;
        Q.push(road(start,0));
        d[start]=0;
        while (!Q.empty())
        {
                road t=Q.top();
                Q.pop();
                int u=t.u;
                for (ul i=0; i<g[u].size(); i++)
                {
                        int v=g[u][i];
                        if (d[u]+cost[u][i]<d[v])
                        {
                                d[v]=d[u]+cost[u][i];
                                parent[v]=u;
                                Q.push(road(v,d[v]));
                        }
                }
        }
        return;
}

int main()
{
        int n, m, road_out, road_cost, cases=1;
        while (scanf("%d", &n) && n)
        {
                for (int i=1; i<=n; i++)
                {
                        cin>>m;
                        for (int j=1; j<=m; j++)
                        {
                                cin>>road_out>>road_cost;
                                g[i].pb(road_out);
                                cost[i].pb(road_cost);
                        }
                }
                scanf("%d%d", &start, &end);
                dijkstra(n);
                //cout<<d[end]<<endl;
                g.clear(); cost.clear();
                int current=end;
                vi path;
                while (current!=start)
                {
                        path.pb(parent[current]);
                        current=parent[current];
                }
                printf("Case %d: Path = ", cases++);
                for (int j=(int)path.size()-1; j>-1; j--)
                        cout<<path[j]<<" ";
                printf("%d; %d second delay\n", end, d[end]);
        }
        return 0;
}
```

5.5 Dominator Tree

```
// Problem: LightOJ Sabotaging Contest
// n - number of cities, m - number of edges, (u,v,t) - edge and cost
// Each of the q lines gives a query of k cities n[1],n[2],...,n[k];
// We have to find the number of nodes where if any one of them is
    removed, the
// shortest path to 0 from n[1]...n[k] will be increased. We also have to
    print
// the number of nodes which will be affected by such removal.

/* Solution
        Run Dijkstra, build shortest path dag, take topsort order and
            reverse it,
        according to the reversed order add one edge at a time to build
            dominator tree
        Finally, run dfs to find the level of each node and subtree size.
            Answer is the
        (level of the lca of the nodes n[1]...n[k] + 1) and subtree size
            of this ancestor
*/
```

```
vi graph[MAX], cost[MAX], dag[MAX], parent[MAX], Tree[MAX];
int u, v, t, n, m;
int dist[MAX];
vector<int> all;
int L[MAX], table[MAX][18], sub[MAX];
bool visited[MAX];

void clear()
{
	FOR(i,0,n)
	{
		graph[i].clear();
		cost[i].clear();
		dag[i].clear();
		parent[i].clear();
		Tree[i].clear();
		sub[i]=0;
	}
	all.clear();
	ms(table,-1);
	ms(visited,false);
}

void dfs(int u)
{
	sub[u]++;

	FOR(j,0,Tree[u].size())
	{
		int v=Tree[u][j];
		dfs(v);
		sub[u]+=sub[v];
	}
}

int query(int p, int q)
{
	if(L[p]<L[q]) swap(p,q);

	int x=1;

	while(true)
	{
		if((1<<(x+1))>L[p])
			break;
			x++;
	}

	FORr(i,x,0)
	{
		if(L[p]-(1<<i) >= L[q])
			p=table[p][i];
	}

	if(p==q) return p;

	FORr(i,x,0)
	{
		if(table[p][i]!=-1 && table[p][i]!=table[q][i])
		{
			p=table[p][i];
			q=table[q][i];
		}
	}

	return table[p][0];
}

void build(int curr)
{
	for(int j=1; (1<<j) < n; j++)
	{
		if(table[curr][j-1]!=-1)
			table[curr][j]=table[table[curr][j-1]][j-1];
	}
}

void dijkstra()
{
	priority_queue<pii,vpii,greater<pii> > PQ;
	PQ.push(pii(0,0));
	FOR(i,0,n) dist[i]=INF;
	dist[0]=0;

	while(!PQ.empty())
	{
		pii t=PQ.top();
		PQ.pop();

		int u=t.second;
```

```
        FOR(j,0,graph[u].size())
        {
            int v=graph[u][j];

            if(dist[u]+cost[u][j]<dist[v])
            {
                dist[v]=dist[u]+cost[u][j];
                PQ.push(pii(dist[v],v));
            }
        }
    }
}

void buildDag()
{
    FOR(i,0,n)
    {
        FOR(j,0,graph[i].size())
        {
            int v=graph[i][j];

            if(dist[i]!=INF && dist[v]!=INF &&
               dist[v]==dist[i]+cost[i][j])
            {
                dag[i].pb(v);
                parent[v].pb(i);
            }
        }
    }
}

void topsort(int u)
{
    visited[u]=true;

    FOR(j,0,dag[u].size())
    {
        if(!visited[dag[u][j]]) topsort(dag[u][j]);
    }

    all.pb(u);
}

void buildTree()
{
    L[0]=0;
    REVERSE(all);

    FOR(i,0,all.size())
    {
        int now=all[i];

        if(parent[now].size())
        {
            int anc=parent[now][0];

            FOR(j,1,parent[now].size())
            {
                anc=query(anc,parent[now][j]);
            }

            L[now]=L[anc]+1;
            table[now][0]=anc;
            Tree[anc].pb(now);

            build(now);
        }
    }
}

int main()
{
  int test, cases=1;

  scanf("%d", &test);

  while(test--)
  {
    scanf("%d%d", &n, &m);

    FOR(i,0,m)
    {
        scanf("%d%d%d", &u, &v, &t);

        graph[u].pb(v);
        graph[v].pb(u);
        cost[u].pb(t);
        cost[v].pb(t);
    }
```

```
        dijkstra();
        buildDag();
        topsort(0);
        buildTree();
        dfs(0);

        int q; scanf("%d", &q);

        printf("Case %d:\n", cases++);

        while(q--)
        {
                int x, u;

                scanf("%d", &x);

                int anc=-1;

                FOR(i,0,x)
                {
                        scanf("%d", &u);

                        if(dist[u]==INF) continue;

                        if(anc==-1) anc=u;
                        else anc=query(anc,u);
                }

                if(anc==-1) printf("0\n");
                else printf("%d %d\n", L[anc]+1, sub[anc]);
        }

        clear();
    }
    return 0;
}
```

5.6 Edmonds Matching

```
/*
 * Algorithm: Edmonds Blossom Maximum Matching in Generel Graph
 * Order : O( N^4 )
 * Note : vertx must be indexing based
 */

#include<stdio.h>
#include<string.h>
using namespace std;
#define MAX_V 103
#define MAX_E MAX_V*MAX_V

long nV,nE,Match[MAX_V];
long Last[MAX_V], Next[MAX_E], To[MAX_E];
long eI;
long q[MAX_V], Pre[MAX_V], Base[MAX_V];
bool Hash[MAX_V], Blossom[MAX_V], Path[MAX_V];

void Insert(long u, long v) {
    To[eI] = v, Next[eI] = Last[u], Last[u] = eI++;
    To[eI] = u, Next[eI] = Last[v], Last[v] = eI++;
}

long Find_Base(long u, long v) {
    memset( Path,0,sizeof(Path));
    for (;;) {
        Path[u] = 1;
        if (Match[u] == -1) break;
        u = Base[Pre[Match[u]]];
    }
    while (Path[v] == 0) v = Base[Pre[Match[v]]];
    return v;
}

void Change_Blossom(long b, long u) {
    while (Base[u] != b) {
        long v = Match[u];
        Blossom[Base[u]] = Blossom[Base[v]] = 1;
        u = Pre[v];
        if (Base[u] != b) Pre[u] = v;
    }
}

long Contract(long u, long v) {
    memset( Blossom,0,sizeof(Blossom));
    long b = Find_Base(Base[u], Base[v]);
    Change_Blossom(b, u);
    Change_Blossom(b, v);
```

```
    if (Base[u] != b) Pre[u] = v;
    if (Base[v] != b) Pre[v] = u;
    return b;
}

void Augment(long u) {
    while (u != -1) {
        long v = Pre[u];
        long k = Match[v];
        Match[u] = v;
        Match[v] = u;
        u = k;
    }
}

long Bfs( long p ){
    memset( Pre,-1,sizeof(Pre));
    memset( Hash,0,sizeof(Hash));
    long i;
    for( i=1;i<=nV;i++ ) Base[i] = i;
    q[1] = p, Hash[p] = 1;
    for (long head=1, rear=1; head<=rear; head++) {
        long u = q[head];
        for (long e=Last[u]; e!=-1; e=Next[e]) {
            long v = To[e];
            if (Base[u]!=Base[v] and v!=Match[u]) {
                if (v==p or (Match[v]!=-1 and Pre[Match[v]]!=-1)) {
                    long b = Contract(u, v);
                    for( i=1;i<=nV;i++ ) if (Blossom[Base[i]]==1) {
                        Base[i] = b;
                        if (!Hash[i]) {
                            Hash[i] = 1;
                            q[++rear] = i;
                        }
                    }
                } else if (Pre[v]==-1) {
                    Pre[v] = u;
                    if (Match[v]==-1) {
                        Augment(v);
                        return 1;
                    }
                    else {
                        q[++rear] = Match[v];
                        Hash[Match[v]] = 1;
                    }
                }
            }
        }
    }
    return 0;
}

long Edmonds_Blossom( void ){
    long i,Ans = 0;
    memset( Match,-1,sizeof(Match));
    for( i=1;i<=nV;i++ ) if (Match[i] == -1) Ans += Bfs(i);
    return Ans;
}


int main( void ){
    eI = 0;
    memset( Last,-1,sizeof(Last));

}
```

5.7 Hopcroft Karp

```
vector< int > graph[MAX];
int n, m, match[MAX], dist[MAX];
int NIL=0;

bool bfs()
{
    int i, u, v, len;
    queue< int > Q;
    for(i=1; i<=n; i++)
    {
        if(match[i]==NIL)
        {
            dist[i] = 0;
            Q.push(i);
        }
        else dist[i] = INF;
    }
    dist[NIL] = INF;
    while(!Q.empty())
    {
```

```
        u = Q.front(); Q.pop();
        if(u!=NIL)
        {
            len = graph[u].size();
            for(i=0; i<len; i++)
            {
                v = graph[u][i];
                if(dist[match[v]]==INF)
                {
                    dist[match[v]] = dist[u] + 1;
                    Q.push(match[v]);
                }
            }
        }
    }
    return (dist[NIL]!=INF);
}

bool dfs(int u)
{
    int i, v, len;
    if(u!=NIL)
    {
        len = graph[u].size();
        for(i=0; i<len; i++)
        {
            v = graph[u][i];
            if(dist[match[v]]==dist[u]+1)
            {
                if(dfs(match[v]))
                {
                    match[v] = u;
                    match[u] = v;
                    return true;
                }
            }
        }
        dist[u] = INF;
        return false;
    }
    return true;
}

int hopcroft_karp()
{
    int matching = 0, i;
    // match[] is assumed NIL for all vertex in graph
    // All nodes on left and right should be distinct
    while(bfs())
        for(i=1; i<=n; i++)
            if(match[i]==NIL && dfs(i))
                matching++;
    return matching;
}

void clear()
{
    FOR(j,0,MAX) graph[j].clear();
    ms(match,NIL);
}

int main()
{
        // ios_base::sync_with_stdio(0);
        // cin.tie(NULL); cout.tie(NULL);
        // freopen("in.txt","r",stdin);

    // SPOJ - Fast Maximum Matching

        int p, x, y;

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

        FOR(i,0,p)
        {
                scanf("%d%d", &x, &y);
                graph[x].pb(n+y);
                graph[n+y].pb(x);
        }

        printf("%d\n", hopcroft_karp());

        return 0;
}
```

5.8 Hungarian Weighted Matching

```
// hungarian weighted matching algo
```

```cpp
// finds the max cost of max matching, to find mincost, add edges as
    negatives

template<typename T>
struct KuhnMunkras { // n for left, m for right
  int n, m, match[maxM];
  T g[maxN][maxM], lx[maxN], ly[maxM], slack[maxM];
  bool vx[maxN], vy[maxM];

  void init(int n_, int m_) {
    MEM(g,0); n = n_, m = m_;
  }

  void add(int u, int v, T w) {
    g[u][v] = w;
  }

  bool find(int x) {
    vx[x] = true;
    for (int y = 1; y <= m; ++y) {
      if (!vy[y]) {
        T delta = lx[x] + ly[y] - g[x][y];
        if (equalT(delta, T(0))) {
          vy[y] = true;
          if (match[y] == 0 || find(match[y])) {
            match[y] = x;
            return true;
          }
        } else slack[y] = min(slack[y], delta);
      }
    }
    return false;
  }

  T matching() { // maximum weight matching
    fill(lx + 1, lx + 1 + n, numeric_limits<T>::lowest());
    MEM(ly,0);
    MEM(match,0);
    for (int i = 1; i <= n; ++i) {
      for (int j = 1; j <= m; ++j) lx[i] = max(lx[i], g[i][j]);
    }
    for (int k = 1; k <= n; ++k) {
      fill(slack + 1, slack + 1 + m, numeric_limits<T>::max());
      while (true) {
        MEM(vx,0);
        MEM(vy,0);
        if (find(k)) break;
        else {
          T delta = numeric_limits<T>::max();
          for (int i = 1; i <= m; ++i) {
            if (!vy[i]) delta = min(delta, slack[i]);
          }
          for (int i = 1; i <= n; ++i) {
            if (vx[i]) lx[i] -= delta;
          }
          for (int i = 1; i <= m; ++i) {
            if (vy[i]) ly[i] += delta;
            if (!vy[i]) slack[i] -= delta;
          }
        }
      }
    }
    T result = 0;
    for (int i = 1; i <= n; ++i) result += lx[i];
    for (int i = 1; i <= m; ++i) result += ly[i];
    return result;
  }
};
```

5.9 Kruskal

```cpp
struct edge
{
	int u, v, w;
	bool operator < (const edge & p) const
	{
		return w < p.w;
	}
};
edge get;
int parent[100];
vector <edge> e;
int find(int r)
{
	if (parent[r] == r)
		return r;
	return parent[r] = find(parent[r]);
}
```

```
int mst(int n)
{
	sort(e.begin(), e.end());
	for (int i = 1; i <= n; i++)
		parent[i] = i;
	int cnt = 0, s = 0;
	for (int i = 0; i < (int)e.size(); i++)
	{
		int u = find(e[i].u);
		int v = find(e[i].v);
		if (u != v)
		{
			parent[u] = v;
			cnt++;
			s += e[i].w;
			if (cnt == n - 1)
				break;
		}
	}
}
```

5.10 LCA

```
vi graph[100];
int P[100], L[100], table[100][20];

void dfs(int from, int to, int depth)
{
	P[to]=from;
	L[to]=depth;
	FOR(i,0,(int)graph[to].size())
	{
		int v=graph[to][i];
		if(v==from)
			continue;
		dfs(to,v,depth+1);
	}
}

int query(int n, int p, int q)
{
	if(L[p]<L[q]) swap(p,q);
	int x=1;

	while(true)
	{
		if((1<<(x+1))>L[p])
			break;
		x++;
	}

	FORr(i,x,0)
	{
		if(L[p]-(1<<i) >= L[q])
			p=table[p][i];
	}

	if(p==q) return p;

	FORr(i,x,0)
	{
		if(table[p][i]!=-1 && table[p][i]!=table[q][i])
		{
			p=table[p][i];
			q=table[q][i];
		}
	}

	return P[p];
}


void build(int n)
{
	ms(table,-1);

	FOR(i,0,n)
		table[i][0]=P[i];

	for(int j=1; 1<<j < n; j++)
	{
		for(int i=0; i<n; i++)
		{
			if(table[i][j-1]!=-1)
				table[i][j]=table[table[i][j-1]][j-1];
		}
	}
```

```
}
```

5.11 Max Flow Dinic 2

```
//
// Dinic's maximum flow
//
// Description:
//   Given a directed network G = (V, E) with edge capacity c: E->R.
//   The algorithm finds a maximum flow.
//
// Algorithm:
//   Dinic's blocking flow algorithm.
//
// Complexity:
//   O(n^2 m), but very fast in practice.
//   In particular, for a unit capacity graph,
//   it runs in O(m min{m^{1/2}, n^{2/3}}).
//
// Verified:
//   SPOJ FASTFLOW
//
// Reference:
//   E. A. Dinic (1970):
//   Algorithm for solution of a problem of maximum flow in networks with
//   power estimation.
//   Soviet Mathematics Doklady, vol. 11, pp. 1277-1280.
//
//   B. H. Korte and J. Vygen (2008):
//   Combinatorial Optimization: Theory and Algorithms.
//   Springer Berlin Heidelberg.
//

#include <iostream>
#include <vector>
#include <cstdio>
#include <queue>
#include <algorithm>
#include <functional>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

const long long INF = (1ll << 50);
struct graph {
	typedef long long flow_type;
	struct edge {
		int src, dst;
		flow_type capacity, flow;
		size_t rev;
	};
	int n;
	vector<vector<edge>> adj;
	graph(int n) : n(n), adj(n) { }
	void add_edge(int src, int dst, flow_type capacity) {
		adj[src].push_back({src, dst, capacity, 0,
		    adj[dst].size()});
		adj[dst].push_back({dst, src, 0, 0, adj[src].size() - 1});
	}
	flow_type max_flow(int s, int t) {
		vector<int> level(n), iter(n);
		function<int(void)> levelize = [&]() { // foward levelize
			level.assign(n, -1); level[s] = 0;
			queue<int> Q; Q.push(s);
			while (!Q.empty()) {
				int u = Q.front(); Q.pop();
				if (u == t) break;
				for (auto &e : adj[u]) {
					if (e.capacity > e.flow &&
					    level[e.dst] < 0) {
					   Q.push(e.dst);
					   level[e.dst] = level[u] + 1;
					}
				}
			}
			return level[t];
		};
		function<flow_type(int, flow_type)> augment = [&](int u,
		    flow_type cur) {
			if (u == t) return cur;
			for (int &i = iter[u]; i < adj[u].size(); ++i) {
				edge &e = adj[u][i], &r = adj[e.dst][e.rev];
				if (e.capacity > e.flow && level[u] <
				    level[e.dst]) {
```

```
                    flow_type f = augment(e.dst, min(cur,
                        e.capacity - e.flow));
                    if (f > 0) {
                        e.flow += f;
                        r.flow -= f;
                        return f;
                    }
                }
            }
            return flow_type(0);
        };
        for (int u = 0; u < n; ++u) // initialize
            for (auto &e : adj[u]) e.flow = 0;

        flow_type flow = 0;
        while (levelize() >= 0) {
            fill(all(iter), 0);
            for (flow_type f; (f = augment(s, INF)) > 0; )
                flow += f;
        }
        return flow;
    }
};

int main() {
    for (int n, m; scanf("%d %d", &n, &m) == 2; ) {
        graph g(n);
        for (int i = 0; i < m; ++i) {
            int u, v, w;
            scanf("%d %d %d", &u, &v, &w);
            //g.add_edge(u, v, w);
            g.add_edge(u - 1, v - 1, w);
        }
        printf("%lld\n", g.max_flow(0, n - 1));
    }
}
```

5.12 Max Flow Dinic

```
/*
Feasible flow in network with upper + lower constraint, no source, no
    sink:
cap' = upper bound - lower bound.
Add source s, sink t.
Let M[v] = (sum of lowerbounds of ingoing edges to v) - (sum of lower
    bounds of outgoing edges from v).
For all v, if M[v] > 0, add (s, v, M), else add (v, t, -M).
If all outgoing edges from S are full --> feasible flow exists, it is
    flow + lower bounds.

Max flow with both upper + lower constraints, source s, sink t: add edge
    (t, s, +INF).
Binary search lower bound, check whether feasible flow exists WITHOUT
    source / sink

*/

struct Edge
{
    int to, rev, f, cap;
};


class Dinic
{
public:

    int dist[MAX], q[MAX], work[MAX], src, dest;
    vector<Edge> graph[MAX];
    // MAX equals to node_number

    void init(int sz)
    {
        FOR(i,0,sz+1) graph[i].clear();
    }

    void clearFlow(int sz)
    {
        FOR(i,0,sz+1)
        {
            FOR(j,0,graph[i].size())
                graph[i][j].f=0;
        }
    }

    void addEdge(int s, int t, int cap)
    {
        Edge a={t,(int)graph[t].size(),0,cap};
```

```
        Edge b={s,(int)graph[s].size(),0,0};

        // If our graph has bidirectional edges
        // Capacity for the Edge b will equal to cap
        // For directed, it is 0

        graph[s].emplace_back(a);
        graph[t].emplace_back(b);
}

bool bfs()
{
        ms(dist,-1);
        dist[src]=0;
        int qt=0;
        q[qt++]=src;

        for(int qh=0; qh<qt; qh++)
        {
                int u=q[qh];

                for(auto &e: graph[u])
                {
                        int v=e.to;

                        if(dist[v]<0 && e.f<e.cap)
                        {
                                dist[v]=dist[u]+1;
                                q[qt++]=v;
                        }
                }
        }

        return dist[dest]>=0;
}

int dfs(int u, int f)
{
        if(u==dest) return f;

        for(int &i=work[u]; i<(int)graph[u].size(); i++)
        {
                Edge &e=graph[u][i];

                if(e.cap<=e.f) continue;

                int v=e.to;

                if(dist[v]==dist[u]+1)
                {
                        int df=dfs(v,min(f,e.cap-e.f));

                        if(df>0)
                        {
                                e.f+=df;
                                graph[v][e.rev].f-=df;

                                return df;
                        }
                }
        }

        return 0;
}

int maxFlow(int _src, int _dest)
{
        src=_src;
        dest=_dest;

        int result=0;

        while(bfs())
        {
                // debug;
                fill(work,work+MAX,0);
                while(int delta=dfs(src,INF))
                        result+=delta;
        }

        return result;
}
};
```

5.13 Max Flow Edmond Karp

```
//
// Maximum Flow (Edmonds-Karp)
```

```cpp
//
// Description:
//   Given a directed network G = (V, E) with edge capacity c: E->R.
//   The algorithm finds a maximum flow.
//
// Algorithm:
//   Edmonds-Karp shortest augmenting path algorithm.
//
// Complexity:
//   O(n m^2)
//
// Verified:
//   AOJ GRL_6_A: Maximum Flow
//
// Reference:
//   B. H. Korte and J. Vygen (2008):
//   Combinatorial Optimization: Theory and Algorithms.
//   Springer Berlin Heidelberg.
//

#include <iostream>
#include <vector>
#include <queue>
#include <cstdio>
#include <algorithm>
#include <functional>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

const int INF = 1 << 30;
struct graph {
	int n;
	struct edge {
		int src, dst;
		int capacity, residue;
		size_t rev;
	};
	edge &rev(edge e) { return adj[e.dst][e.rev]; };

	vector<vector<edge>> adj;
	graph(int n) : n(n), adj(n) { }
	void add_edge(int src, int dst, int capacity) {
		adj[src].push_back({src, dst, capacity, 0,
		    adj[dst].size()});
		adj[dst].push_back({dst, src, 0, 0, adj[src].size() - 1});
	}
	int max_flow(int s, int t) {
		for (int u = 0; u < n; ++u)
			for (auto &e : adj[u]) e.residue = e.capacity;
		int total = 0;
		while (1) {
			vector<int> prev(n, -1); prev[s] = -2;
			queue<int> que; que.push(s);
			while (!que.empty() && prev[t] == -1) {
				int u = que.front(); que.pop();
				for (edge &e : adj[u]) {
					if (prev[e.dst] == -1 && e.residue >
					    0) {
						prev[e.dst] = e.rev;
						que.push(e.dst);
					}
				}
			}
			if (prev[t] == -1) break;
			int inc = INF;
			for (int u = t; u != s; u = adj[u][prev[u]].dst)
				inc = min(inc, rev(adj[u][prev[u]]).residue);
			for (int u = t; u != s; u = adj[u][prev[u]].dst) {
				adj[u][prev[u]].residue += inc;
				rev(adj[u][prev[u]]).residue -= inc;
			}
			total += inc;
		} // { u : visited[u] == true } is s-side
		return total;
	}
};

int main() {
	for (int n, m; scanf("%d %d", &n, &m) == 2; ) {
		graph g(n);
		for (int i = 0; i < m; ++i) {
			int u, v, w;
			scanf("%d %d %d", &u, &v, &w);
			g.add_edge(u, v, w);
		}
		printf("%d\n", g.max_flow(0, n - 1));
```

```
        }
}
```

5.14 Max Flow Ford Fulkerson

```
//
// Ford-Fulkerson's maximum flow
//
// Description:
//   Given a directed network G = (V, E) with edge capacity c: E->R.
//   The algorithm finds a maximum flow.
//
// Algorithm:
//   Ford-Fulkerson's augmenting path algorithm
//
// Complexity:
//   O(m F), where F is the maximum flow value.
//
// Verified:
//   AOJ GRL_6_A: Maximum Flow
//
// Reference:
//   B. H. Korte and J. Vygen (2008):
//   Combinatorial Optimization: Theory and Algorithms.
//   Springer Berlin Heidelberg.
//

#include <iostream>
#include <vector>
#include <cstdio>
#include <algorithm>
#include <functional>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

const int INF = 1 << 30;
struct graph {
        typedef long long flow_type;
        struct edge {
                int src, dst;
                flow_type capacity, flow;
                size_t rev;
        };
        int n;
        vector<vector<edge>> adj;
        graph(int n) : n(n), adj(n) { }
        void add_edge(int src, int dst, flow_type capacity) {
                adj[src].push_back({src, dst, capacity, 0,
                    adj[dst].size()});
                adj[dst].push_back({dst, src, 0, 0, adj[src].size() - 1});
        }
        int max_flow(int s, int t) {
                vector<bool> visited(n);
                function<flow_type(int, flow_type)> augment = [&](int u,
                    flow_type cur) {
                        if (u == t) return cur;
                        visited[u] = true;
                        for (auto &e : adj[u]) {
                                if (!visited[e.dst] && e.capacity > e.flow) {
                                        flow_type f = augment(e.dst,
                                            min(e.capacity - e.flow, cur));
                                        if (f > 0) {
                                                e.flow += f;
                                                adj[e.dst][e.rev].flow -= f;
                                                return f;
                                        }
                                }
                        }
                        return flow_type(0);
                };
                for (int u = 0; u < n; ++u)
                        for (auto &e : adj[u]) e.flow = 0;

                flow_type flow = 0;
                while (1) {
                        fill(all(visited), false);
                        flow_type f = augment(s, INF);
                        if (f == 0) break;
                        flow += f;
                }
                return flow;
        }
};
```

```
int main() {
	for (int n, m; scanf("%d %d", &n, &m) == 2; ) {
		graph g(n);
		for (int i = 0; i < m; ++i) {
			int u, v, w;
			scanf("%d %d %d", &u, &v, &w);
			g.add_edge(u, v, w);
		}
		printf("%d\n", g.max_flow(0, n - 1));
	}
}
```

5.15 Max Flow Goldberg Tarjan

```
//
// Maximum Flow (Goldberg-Tarjan, aka. Push-Relabel, Preflow-Push)
//
// Description:
//   Given a directed network G = (V, E) with edge capacity c: E->R.
//   The algorithm finds a maximum flow.
//
// Algorithm:
//   Goldberg-Tarjan's push-relabel algorithm with gap-heuristics.
//
// Complexity:
//   O(n^3)
//
// Verified:
//   SPOJ FASTFLOW
//
// Reference:
//   B. H. Korte and Jens Vygen (2008):
//   Combinatorial Optimization: Theory and Algorithms.
//   Springer Berlin Heidelberg.
//

#include <iostream>
#include <vector>
#include <cstdio>
#include <queue>
#include <algorithm>
#include <functional>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

const long long INF = (1ll << 50);
struct graph {
	typedef long long flow_type;
	struct edge {
		int src, dst;
		flow_type capacity, flow;
		size_t rev;
	};
	int n;
	vector<vector<edge>> adj;
	graph(int n) : n(n), adj(n) { }

	void add_edge(int src, int dst, int capacity) {
		adj[src].push_back({src, dst, capacity, 0,
		    adj[dst].size()});
		adj[dst].push_back({dst, src, 0, 0, adj[src].size() - 1});
	}

	flow_type max_flow(int s, int t) {
		vector<flow_type> excess(n);
		vector<int> dist(n), active(n), count(2 * n);
		queue<int> Q;
		auto enqueue = [&](int v) {
			if (!active[v] && excess[v] > 0) { active[v] =
			    true; Q.push(v); }
		};
		auto push = [&](edge & e) {
			flow_type f = min(excess[e.src], e.capacity -
			    e.flow);
			if (dist[e.src] <= dist[e.dst] || f == 0) return;
			e.flow += f;
			adj[e.dst][e.rev].flow -= f;
			excess[e.dst] += f;
			excess[e.src] -= f;
			enqueue(e.dst);
		};

		dist[s] = n; active[s] = active[t] = true;
		count[0] = n - 1; count[n] = 1;
```

```
        for (int u = 0; u < n; ++u)
            for (auto &e : adj[u]) e.flow = 0;
        for (auto &e : adj[s]) {
            excess[s] += e.capacity;
            push(e);
        }
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            active[u] = false;

            for (auto &e : adj[u]) push(e);
            if (excess[u] > 0) {
                if (count[dist[u]] == 1) {
                    int k = dist[u]; // Gap Heuristics
                    for (int v = 0; v < n; v++) {
                        if (dist[v] < k) continue;
                        count[dist[v]]--;
                        dist[v] = max(dist[v], n + 1);
                        count[dist[v]]++;
                        enqueue(v);
                    }
                } else {
                    count[dist[u]]--; // Relabel
                    dist[u] = 2 * n;
                    for (auto &e : adj[u])
                        if (e.capacity > e.flow)
                            dist[u] = min(dist[u],
                              dist[e.dst] + 1);
                    count[dist[u]]++;
                    enqueue(u);
                }
            }
        }

        flow_type flow = 0;
        for (auto e : adj[s]) flow += e.flow;
        return flow;
    }
};

int main() {
    for (int n, m; scanf("%d %d", &n, &m) == 2; ) {
        graph g(n);
        for (int i = 0; i < m; ++i) {
            int u, v, w;
            scanf("%d %d %d", &u, &v, &w);
            g.add_edge(u, v, w);
        }
        printf("%d\n", g.max_flow(0, n - 1));
    }
}
```

5.16 Maximum Bipartite Matching and Min Vertex Cover

```
int n, m, p; // n = # of nodes on left, m = # of nodes on right
vi bp[N]; // bipartite graph
int matched[N], revmatch[N];
bool seen[N], visited[2][N];

bool trymatch(int u)
{
    FOR(j,0,bp[u].size())
    {
        int v=bp[u][j];
        if(seen[v]) continue;

        seen[v]=true;

        // v is on right, u on left
        if(matched[v]<0 || trymatch(matched[v]))
        {
            matched[v]=u;
            revmatch[u]=v;
            return true;
        }
    }

    return false;
}

// 0 based
int maxbpm(int sz)
{
    ms(matched,-1);
    ms(revmatch,-1); // for min-vertex-cover

    int ret=0;
```

```
        FOR(i,0,sz)
        {
                ms(seen,false);
                if(trymatch(i)) ret++;
        }

        return ret;
}

void dfsLast(int u, bool side)
{
        if(visited[side][u]) return;
        visited[side][u]=true;

        if(!side)
        {
                for(int i=0; i<n; i++)
                {
                        if(graph[u][i] && matched[u]!=i)
                                dfsLast(i,1-side);
                }
        }
        else dfsLast(matched[u],1-side);
}

void findMinVertexCover()
{
        FOR(i,0,n)
        {
                if(revmatch[i]==-1)
                {
                        dfsLast(i,0);
                }
        }
        // Assuming both sides have n nodes
        vi mvc, mis; // min vertex cover, max independent set
        FOR(i,0,n)
        {
                if(!visited[0][i] || visited[1][i]) mvc.pb(i);
                if(!(!visited[0][i] || visited[1][i])) mis.pb(i);
        }
}
```

5.17 Min Cost Arborescence

```
// Min Cost Arboroscense class in C++
// Directed MST
// dir_mst returns the cost O(EV)?

struct Edge {
    int u, v;
    ll dist;
    int kbps;
};

struct MinCostArborescence{
    int n, m;
    Edge allEdges[MAX];
    int done[62], prev[62], id[62];
    ll in[62];

    void init(int n)
    {
        this->n = n;
        m = 0;
    }

    void add_Edge(int u, int v, ll dist)
    {
        allEdges[m++] = {u,v,dist,0};
    }

    void add_Edge(Edge e)
    {
        allEdges[m++] = e;
    }

    ll dir_mst(int root) {
        ll ans = 0;
        while (true) {
            for (int i = 0; i < n; i++) in[i] = INF;
            for (int i = 0; i < m; i++) {
                int u = allEdges[i].u;
                int v = allEdges[i].v;
                if (allEdges[i].dist < in[v] && u != v) {
                    in[v] = allEdges[i].dist;
                    prev[v] = u;
                }
```

```
        }

        for (int i = 0; i < n; i++) {
            if (i == root) continue;
            if (in[i] == INF) return -1;
        }

        int cnt = 0;
        memset(id, -1, sizeof(id));
        memset(done, -1, sizeof(done));
        in[root] = 0;

        for (int i = 0; i < n; i++)
        {
            ans += in[i];
            int v = i;
            while (done[v] != i && id[v] == -1 && v != root) {
                done[v] = i;
                v = prev[v];
            }
            if (v != root && id[v] == -1) {
                for (int u = prev[v]; u != v; u = prev[u])
                    id[u] = cnt;
                id[v] = cnt++;
            }
        }
        if (cnt == 0) break;
        for (int i = 0; i < n; i++)
            if (id[i] == -1) id[i] = cnt++;
        for (int i = 0; i < m; i++) {
            int v = allEdges[i].v;
            allEdges[i].u = id[allEdges[i].u];
            allEdges[i].v = id[allEdges[i].v];
            if (allEdges[i].u != allEdges[i].v)
                allEdges[i].dist -= in[v];
        }
        n = cnt;
        root = id[root];
    }
    return ans;
  }
} Arboroscense;
```

5.18 Min Cost Max Flow 1

```
//
// Minimum Cost Maximum Flow (Tomizawa, Edmonds-Karp's successive
    shortest path)
//
// Description:
//   Given a directed graph G = (V,E) with nonnegative capacity c and
    cost w.
//   The algorithm find a maximum s-t flow of G with minimum cost.
//
// Algorithm:
//   Tomizawa (1971), and Edmonds and Karp (1972)'s
//   successive shortest path algorithm,
//   which is also known as the primal-dual method.
//
// Complexity:
//   O(F m log n), where F is the amount of maximum flow.


// Caution: Probably does not support Negative Costs
// Negative cost is supported in an implementation named:
    mincostmaxflow2.cpp


#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())
#define TEST(s) if (!(s)) { cout << __LINE__ << " " << #s << endl;
    exit(-1); }

const long long INF = 1e9;
struct graph {
  typedef int flow_type;
  typedef int cost_type;
  struct edge {
    int src, dst;
    flow_type capacity, flow;
    cost_type cost;
    size_t rev;
  };
  vector<edge> edges;
  void add_edge(int src, int dst, flow_type cap, cost_type cost) {
    adj[src].push_back({src, dst, cap, 0, cost, adj[dst].size()});
```

```
  adj[dst].push_back({dst, src, 0, 0, -cost, adj[src].size()-1});
}
int n;
vector<vector<edge>> adj;
graph(int n) : n(n), adj(n) { }

pair<flow_type, cost_type> min_cost_max_flow(int s, int t) {
  flow_type flow = 0;
  cost_type cost = 0;

  for (int u = 0; u < n; ++u) // initialize
    for (auto &e: adj[u]) e.flow = 0;

  vector<cost_type> p(n, 0);

  auto rcost = [&](edge e) { return e.cost + p[e.src] - p[e.dst]; };
  for (int iter = 0; ; ++iter) {
    vector<int> prev(n, -1); prev[s] = 0;
    vector<cost_type> dist(n, INF); dist[s] = 0;
    if (iter == 0) { // use Bellman-Ford to remove negative cost edges
      vector<int> count(n); count[s] = 1;
      queue<int> que;
      for (que.push(s); !que.empty(); ) {
        int u = que.front(); que.pop();
        count[u] = -count[u];
        for (auto &e: adj[u]) {
          if (e.capacity > e.flow && dist[e.dst] > dist[e.src] +
              rcost(e)) {
            dist[e.dst] = dist[e.src] + rcost(e);
            prev[e.dst] = e.rev;
            if (count[e.dst] <= 0) {
              count[e.dst] = -count[e.dst] + 1;
              que.push(e.dst);
            }
          }
        }
      }
    } else { // use Dijkstra
      typedef pair<cost_type, int> node;
      priority_queue<node, vector<node>, greater<node>> que;
      que.push({0, s});
      while (!que.empty()) {
        node a = que.top(); que.pop();
        if (a.snd == t) break;
        if (dist[a.snd] > a.fst) continue;
        for (auto e: adj[a.snd]) {
          if (e.capacity > e.flow && dist[e.dst] > a.fst + rcost(e)) {
            dist[e.dst] = dist[e.src] + rcost(e);
            prev[e.dst] = e.rev;
            que.push({dist[e.dst], e.dst});
          }
        }
      }
    }
    if (prev[t] == -1) break;

    for (int u = 0; u < n; ++u)
      if (dist[u] < dist[t]) p[u] += dist[u] - dist[t];

    function<flow_type(int,flow_type)> augment = [&](int u, flow_type
        cur) {
      if (u == s) return cur;
      edge &r = adj[u][prev[u]], &e = adj[r.dst][r.rev];
      flow_type f = augment(e.src, min(e.capacity - e.flow, cur));
      e.flow += f; r.flow -= f;
      return f;
    };
    flow_type f = augment(t, INF);
    flow += f;
    cost += f * (p[t] - p[s]);
  }
  return {flow, cost};
}
};
```

5.19 Min Cost Max Flow 2

```
// By zscoder
// From problem: CF Anti Palindromize - 884F
// Thank you ZS.
// Works as max-cost-max-flow if the costs are considered negative
// Slower due to SPFA in some cases?

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

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

    void addEdge(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)
            addEdge(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; ll 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; ll 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;
    }
};
```

5.20 Min Cost Max Flow 3

```
// This gave AC for CF 813D Two Melodies but the other one was TLE
// By sgtlaugh

namespace mcmf{
    const int MAX = 1000010;
    const int INF = 1 << 25;

    int cap[MAX], flow[MAX], cost[MAX], dis[MAX];
    int n, m, s, t, Q[10000010], adj[MAX], link[MAX], last[MAX],
         from[MAX], visited[MAX];

    void init(int nodes, int source, int sink){
        m = 0, n = nodes, s = source, t = sink;
        for (int i = 0; i <= n; i++) last[i] = -1;
    }

    void addEdge(int u, int v, int c, int w){
        adj[m] = v, cap[m] = c, flow[m] = 0, cost[m] = +w, link[m] =
             last[u], last[u] = m++;
        adj[m] = u, cap[m] = 0, flow[m] = 0, cost[m] = -w, link[m] =
             last[v], last[v] = m++;
    }

    bool spfa(){
        int i, j, x, f = 0, l = 0;
        for (i = 0; i <= n; i++) visited[i] = 0, dis[i] = INF;

        dis[s] = 0, Q[l++] = s;
        while (f < l){
            i = Q[f++];
            for (j = last[i]; j != -1; j = link[j]){
                if (flow[j] < cap[j]){
                    x = adj[j];
                    if (dis[x] > dis[i] + cost[j]){
                        dis[x] = dis[i] + cost[j], from[x] = j;
                        if (!visited[x]){
                            visited[x] = 1;
                            if (f && rand() & 7) Q[--f] = x;
                            else Q[l++] = x;
                        }
                    }
                }
            }
            visited[i] = 0;
        }
        return (dis[t] != INF);
    }

    pair <int, int> solve(){
        int i, j;
        int mincost = 0, maxflow = 0;

        while (spfa()){
            int aug = INF;
            for (i = t, j = from[i]; i != s; i = adj[j ^ 1], j = from[i]){
                aug = min(aug, cap[j] - flow[j]);
            }
            for (i = t, j = from[i]; i != s; i = adj[j ^ 1], j = from[i]){
                flow[j] += aug, flow[j ^ 1] -= aug;
            }
            maxflow += aug, mincost += aug * dis[t];
        }
        return make_pair(mincost, maxflow);
    }
}
```

5.21 Prim MST

```
vector <ll> graph[10003], cost[10003];
bool visited[10003];
ll d[10003];
int n, m;

int minKey()
{
        ll mini=INF;
        int minidx;
        for (int i=1; i<=n; i++)
        {
```

```
                if (!visited[i] && d[i]<mini)
                        mini=d[i], minidx=i;
        }
        return minidx;
}

ll Prim()
{
        FOR(i,0,10003)
        {
                d[i]=INF;
                visited[i]=false;
        }
        d[1]=0;
        for (int i=1; i<=n-1; i++)
        {
                int u=minKey();
                visited[u]=true;
                FOR(j,0,graph[u].size())
                {
                        int v=graph[u][j];
                        if(!visited[v] && cost[u][j]<d[v])
                                d[v]=cost[u][j];
                }

        }
        ll ret=0;
        FOR(j,1,n+1)
        {
                // cout<<d[j]<<endl;
                if(d[j]!=INF)
                        ret+=d[j];
        }
        return ret;
}

int main()
{
        int a, b, c;
        scanf("%d%d", &n, &m);
        FOR(i,0,m)
        {
                scanf("%d%d%d", &a, &b, &c);
                graph[a].pb(b);
                graph[b].pb(a);
                cost[a].pb(c);
                cost[b].pb(c);
        }
        cout<<Prim()<<endl;

    return 0;
}
```

5.22 Push Relabel

```
#define sz(x) (int)(x).size()

struct Edge {
    int v;
    ll flow, C;
    int rev;
};

template <int SZ> struct PushRelabel {
    vector<Edge> adj[SZ];
    ll excess[SZ];
    int dist[SZ], count[SZ+1], b = 0;
    bool active[SZ];
    vi B[SZ];

    void addEdge(int u, int v, ll C) {
        Edge a{v, 0, C, sz(adj[v])};
        Edge b{u, 0, 0, sz(adj[u])};
        adj[u].pb(a), adj[v].pb(b);
    }

    void enqueue (int v) {
        if (!active[v] && excess[v] > 0 && dist[v] < SZ) {
            active[v] = 1;
            B[dist[v]].pb(v);
            b = max(b, dist[v]);
        }
    }

    void push (int v, Edge &e) {
        ll amt = min(excess[v], e.C-e.flow);
        if (dist[v] == dist[e.v]+1 && amt > 0) {
            e.flow += amt, adj[e.v][e.rev].flow -= amt;
```

```
            excess[e.v] += amt, excess[v] -= amt;
            enqueue(e.v);
        }
    }

    void gap (int k) {
        FOR(v,1,SZ+1) if (dist[v] >= k) {
            count[dist[v]] --;
            dist[v] = SZ;
            count[dist[v]] ++;
            enqueue(v);
        }
    }

    void relabel (int v) {
        count[dist[v]] --; dist[v] = SZ;
        for (auto e: adj[v]) if (e.C > e.flow) dist[v] = min(dist[v],
             dist[e.v] + 1);
        count[dist[v]] ++;
        enqueue(v);
    }

    void discharge(int v) {
        for (auto &e: adj[v]) {
            if (excess[v] > 0) push(v,e);
            else break;
        }
        if (excess[v] > 0) {
            if (count[dist[v]] == 1) gap(dist[v]);
            else relabel(v);
        }
    }

    ll maxFlow (int s, int t) {
        for (auto &e: adj[s]) excess[s] += e.C;

        count[0] = SZ;
        enqueue(s); active[t] = 1;

        while (b >= 0) {
            if (sz(B[b])) {
                int v = B[b].back(); B[b].pop_back();
                active[v] = 0; discharge(v);
            } else b--;
        }
        return excess[t];
    }
};

PushRelabel<50000> network;
```

5.23 SCC Kosaraju

```
// Kosaraju's strongly connected component
//
// Description:
//   For a graph G = (V, E), u and v are strongly connected if
//   there are paths u -> v and v -> u. This defines an equivalent
//   relation, and its equivalent class is called a strongly
//   connected component.
//
// Algorithm:
//   Kosaraju's algorithm performs DFS on G and rev(G).
//   First DFS finds topological ordering of SCCs, and
//   the second DFS extracts components.
//
// Complexity:
//   O(n + m)
//
// Verified:
//   SPOJ 6818
//
// References:
//   A. V. Aho, J. E. Hopcroft, and J. D. Ullman (1983):
//   Data Structures and Algorithms,
//   Addison-Wesley.
//
#include <iostream>
#include <vector>
#include <cstdio>
#include <cstdlib>
#include <map>
#include <set>
#include <cmath>
#include <cstring>
#include <functional>
#include <algorithm>
```

```
#include <unordered_map>
#include <unordered_set>

using namespace std;

#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())


struct graph {
  int n;
  vector<vector<int>> adj, rdj;
  graph(int n) : n(n), adj(n), rdj(n) { }
  void add_edge(int src, int dst) {
    adj[src].push_back(dst);
    rdj[dst].push_back(src);
  }

  vector<vector<int>> strongly_connected_components() { // kosaraju
    vector<int> ord, visited(n);
    vector<vector<int>> scc;
    function<void(int, vector<vector<int>>&, vector<int>&)> dfs
    = [&](int u, vector<vector<int>> &adj, vector<int> &out) {
      visited[u] = true;
      for (int v : adj[u])
        if (!visited[v]) dfs(v, adj, out);
      out.push_back(u);
    };
    for (int u = 0; u < n; ++u)
      if (!visited[u]) dfs(u, adj, ord);
    fill(all(visited), false);
    for (int i = n - 1; i >= 0; --i)
      if (!visited[ord[i]])
        scc.push_back({}), dfs(ord[i], rdj, scc.back());
    return scc;
  }
};

int main() {
  int n, m;
  scanf("%d %d", &n, &m);
  graph g(n);
  for (int k = 0; k < m; ++k) {
    int i, j;
    scanf("%d %d", &i, &j);
    g.add_edge(i - 1, j - 1);
  }

  vector<vector<int>> scc = g.strongly_connected_components();
  vector<int> outdeg(scc.size());
  vector<int> id(n);
  for (int i = 0; i < scc.size(); ++i)
    for (int u : scc[i]) id[u] = i;
  for (int u = 0; u < n; ++u)
    for (int v : g.adj[u])
      if (id[u] != id[v]) ++outdeg[id[u]];

  if (count(all(outdeg), 0) != 1) {
    printf("0\n");
  } else {
    int i = find(all(outdeg), 0) - outdeg.begin();
    sort(all(scc[i]));
    printf("%d\n%d", scc[i].size(), scc[i][0] + 1);
    for (int j = 1; j < scc[i].size(); ++j)
      printf(" %d", scc[i][j] + 1);
    printf("\n");
  }
}
```

5.24 SCC Tarjan

```
stack<int> st;
vector<vector<int> > scc;
int low[MAX], disc[MAX], comp[MAX];
int dfs_time;
bool in_stack[MAX];

vi graph[MAX];
int n; // node count indexed from 1

void dfs(int u)
{
        low[u] = dfs_time;
        disc[u] = dfs_time;
        dfs_time++;

        in_stack[u] = true;
```

```
        st.push(u);

        int sz = graph[u].size(), v;
        for(int i = 0; i < sz; i++)
        {
                v = graph[u][i];

                if(disc[v] == -1)
                {
                        dfs(v);
                        low[u] = min(low[u], low[v]);
                }
                else if(in_stack[v] == true)
                        low[u] = min(low[u], disc[v]);
        }

        if(low[u] == disc[u])
        {
                scc.push_back(vector<int>());
                while(st.top() != u)
                {
                        scc[scc.size() - 1].push_back(st.top());
                        in_stack[st.top()] = false;
                        st.pop();
                }

                scc[scc.size() - 1].push_back(u);
                in_stack[u] = false;
                st.pop();
        }
}

int tarjan()
{
        memset(comp, -1, sizeof(comp));
        memset(disc, -1, sizeof(disc));
        memset(low, -1, sizeof(low));
        memset(in_stack, 0, sizeof(in_stack));
        dfs_time = 0;

        while(!st.empty())
                st.pop();

        for(int i = 1; i <= n; i++)
                if(disc[i] == -1)
                        dfs(i);

        int sz = scc.size();
        for(int i = 0; i < sz; i++)
                for(int j = 0; j < (int)scc[i].size(); j++)
                        comp[scc[i][j]] = i;

        return sz;
}
```

5.25 kth Shortest Path Length

```
int n, m, x, y, k, a, b, c;
vi Graph[103], Cost[103];
vector<priority_queue<int> > d(103);
priority_queue < pii > Q;

void goDijkstra()
{

        // Here, elements are sorted in decreasing order of the first
            elements
        // of the pairs and then the second elements if equal first
            element.

        // d[i] is the priority_queue of the node i where the best k path
            length
        // will be stored in decreasing order. So, d[i].top() has the
            longest of the
        // first k shortest path.

        d[x].push(0);
        Q.push(MP(x,0));
        // Q contains the nodes in the increasing order of their cost
        // Since the priority_queue sorts the pairs in decreasing order of
            their
        // first element and then second element, to sort it in increasing
            order
        // we will negate the cost and push it.

        while(!Q.empty())
        {
                pii t=Q.top(); Q.pop();
```

```
        int u=t.first, costU=-t.second;
        // Since the actual cost was negated.

        FOR(j,0,Graph[u].size())
        {
                int v=Graph[u][j];

                // prnt(v); prnt(d[v].size());

                // Have we already got k shortest paths? Or is the
                    longest path can be made better?
                if(d[v].size()<k || d[v].top()>costU+Cost[u][j])
                {
                        int temp=costU+Cost[u][j];
                        d[v].push(temp);
                        Q.push(MP(v,-temp));
                }
                if(d[v].size()>k) d[v].pop();
// If we have more than k shortest path for the current node, we
    can pop
// the worst ones.
        }
    }

    if(d[y].size()<k) prnt(-1);
    // We have not found k shortest path for our destination.
    else prnt(d[y].top());
}

int main()
{
    // ios_base::sync_with_stdio(0);
    // cin.tie(NULL); cout.tie(NULL);
    // freopen("in.txt","r",stdin);

    while(scanf("%d%d", &n, &m) && n+m)
    {
        scanf("%d%d%d", &x, &y, &k);

        FOR(i,0,m)
        {
            scanf("%d%d%d", &a, &b, &c);

            Graph[a].pb(b);
            Cost[a].pb(c);
        }

        goDijkstra();

        FOR(i,0,103) Graph[i].clear(), Cost[i].clear();
        FOR(i,0,103)
        {
            while(!d[i].empty()) d[i].pop();
        }

        while(!Q.empty()) Q.pop();
    }


    return 0;
}
```

6 Math

6.1 CRT Diophantine

```
// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
// Return (z,M). On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
    int s, t;
    int d = extended_euclid(x, y, s, t);
    if (a % d != b % d) return make_pair(0, -1);
    return make_pair(mod(s * b * x + t * a * y, x * y) / d, x * y / d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i. Note that the solution is
// unique modulo M = lcm_i (x[i]). Return (z,M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.

PII chinese_remainder_theorem(const VI &x, const VI &a) {
    PII ret = make_pair(a[0], x[0]);
    for (int i = 1; i < x.size(); i++) {
        ret = chinese_remainder_theorem(ret.second, ret.first,
            x[i], a[i]);
```

```
            if (ret.second == -1) break;
        }
        return ret;
}

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
        int d = gcd(a, b);
        if (c % d) {
                x = y = -1;
        } else {
                x = c / d * mod_inverse(a / d, b / d);
                y = (c - a * x) / b;
        }
}
```

6.2 Euler Phi

```
int phi[MAX];
void phi()
{
        for (int i = 1; i < MAX; i++) phi[i] = i;
        for (int i = 2; i < MAX; i++)
        {
                if (phi[i] == i)
                {
                        for (int j = i; j < MAX; j += i)
                        {
                                phi[j] /= i;
                                phi[j] *= (i - 1);
                        }
                }
        }
}
```

6.3 FFT 1

```
/*
FFT is used for fast multiplication.
Main Functionality : mult(a,b)
Parameter : vector<int>a, vector<int> b (Representing a polynomial where
    a[3]
contains co-efficient of the polynomial of degree 3)
Output : The polynomial a*b
*/

typedef complex<double> Complex;

void fft(vector<Complex> & a, bool inv)
{
    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)
    {
        double ang = 2 * PI / len * (inv ? -1 : 1);
        Complex wlen(cos(ang), sin(ang));
        for (int i = 0; i<n; i += len)
        {
            Complex w(1);
            for (int j = 0; j<len / 2; ++j)
            {
                Complex u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }
    if(inv)
    for (int i = 0; i<n; ++i)
        a[i] /= n;

}

vector<int> mult(vector<int>& a, vector<int>& b)
```

```
{
    vector<Complex> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    size_t n = 1;
    while (n < max(a.size(), b.size())) n <<= 1;
    n <<= 1;
    fa.resize(n), fb.resize(n);

    fft(fa, false), fft(fb, false);
    for (size_t i = 0; i<n; ++i)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<int> res;
    res.resize(n);
    for (size_t i = 0; i<n; ++i)
        res[i] = int(fa[i].real() + 0.5);
    return res;
}


vector<int> squ(vector<int>& a)
{
    vector<Complex> fa(a.begin(), a.end()), fb(a.begin(), a.end());
    size_t n = 1;
    while (n < a.size()) n <<= 1;
    n <<= 1;
    fa.resize(n), fb.resize(n);

    fft(fa, false); fb = fa;
    for (size_t i = 0; i < n; ++i)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<int> res;
    res.resize(n);
    for (size_t i = 0; i < n; ++i)
        res[i] = int(fa[i].real() + 0.5);
    return res;
}
```

6.4 FFT 2

```
typedef vector<long long> VL;
typedef complex<long double> CN;

void FFT( vector<CN> &a, bool invert ){
    long i,j,n = a.size();
    for( i=1,j=0;i<n;i++ ){
                long bit = n >> 1 ;
                for( ;j>=bit;bit>>=1 ) j -= bit ;
                j += bit ;
                if( i < j ) swap( a[i],a[j] );
        }
        long len;
        for( len=2;len<=n;len<<=1 ){
            double ang = 2*PI / len * ( invert ? -1:1 );
            CN wlen( cos( ang ) , sin( ang ) );
                for( i=0;i<n;i+=len ){
                        CN w( 1 ) ;
                        for( j=0;j<len/2;j++ ){
                                CN u = a[i+j] , v = a[i+j+len/2]*w;
                                a[i+j] = u+v; a[i+j+len/2] = u-v; w *= wlen;
                        }
                }
        }
        if( invert ){
            for( i=0;i<n;i++ ) a[i] /= n;
        }
}

void Multiply( const VL &a, const VL &b, VL &res ){
    vector<CN> fa( a.begin(),a.end() ), fb( b.begin(),b.end() );
        long i,n = 1 ;
        while( n < max( a.size(),b.size()) ) n <<= 1;
        n <<= 1;
        fa.resize( n ),fb.resize( n );
        FFT( fa,false ) , FFT( fb,false );
        for( i=0;i<n;i++ ) fa[i] *= fb[i];
        FFT( fa,true );
        res.resize( n );
        for( i=0;i<n;i++ ) res[i] = long( fa[i].real() + 0.5 );
        /* if multiplication between 2 number then res need to be mod by
            10 */
}
```

6.5 Gauss Elimination Equations Mod Number Solutions

```
ll pow(ll base, ll p, ll MOD)
{
        if(p == 0) return 1;
        if(p % 2 == 0) { ll d = pow(base, p / 2, MOD); return (d * d) %
             MOD; }
        return (pow(base, p - 1, MOD) * base) % MOD;
}

ll inv(ll x, ll MOD) { return pow(x, MOD - 2, MOD); }

// If MOD equals 2, it becomes XOR operation and we can use vector of
     bitsets to build equation
// Complexity becomes 1/32

ll gauss(vector<vector<ll> > &a, ll MOD)
{
        int n = a.size(), m = a[0].size() - 1;

        for(int i = 0; i < n; i++)
                for(int j = 0; j <= m; j++)
                        a[i][j] = (a[i][j] % MOD + MOD) % MOD;

        vector<int> where(m, -1);
        for(int col = 0, row = 0; col < m && row < n; col++)
    {
        int sel = row;
        for(int i = row; i < n; i++)
                if(a[i][col] > a[sel][col])
                        sel = i;

                if(a[sel][col] == 0) { where[col] = -1; continue; }

        for(int i = col; i <= m; i++)
                        swap(a[sel][i], a[row][i]);
                where[col] = row;

                ll c_inv = inv(a[row][col], MOD);
                for(int i = 0; i < n; i++)
                        if(i != row)
                        {
                                if(a[i][col] == 0) continue;
                ll c = (a[i][col] * c_inv) % MOD;
                for(int j = 0; j <= m; j++)
                                a[i][j] = (a[i][j] - c * a[row][j] % MOD + MOD) % MOD;
                        }

                        row++;
        }

        vector<ll> ans(m, 0);
        ll result = 1;

        // for counting rank, take the count of where[i]==-1
        for(int i = 0; i < m; i++)
            if(where[i] != -1) ans[i] = (a[where[i]][m] * inv(a[where[i]][i],
                MOD)) % MOD;
                        else result = (result * MOD) % mod;

            // This is validity check probably.
            // May not be needed

        for(int i = 0; i < n; i++)
            {
                        ll sum = a[i][m] % MOD;
                        for(int j = 0; j < m; j++)
                                sum = (sum + MOD - (ans[j] * a[i][j]) % MOD) % MOD;

                        if(sum != 0) return 0;
            }

            return result;
}
```

6.6 Gauss Jordan Elimination

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:   a[][] = an nxn matrix
//          b[][] = an nxm matrix
```

```
//
// OUTPUT:  X      = an nxm matrix (stored in b[][])
//          A^{-1} = an nxn matrix (stored in a[][])
//          returns determinant of a[][]

// Example used: LightOJ Snakes and Ladders

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

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 = 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;
            }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl;
        exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; 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[pk][q] * c;
      for (int q = 0; q < m; 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;
}

int main() {
  const int n = 4;
  const int m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++) {
    a[i] = VT(A[i], A[i] + n);
    b[i] = VT(B[i], B[i] + m);
  }

  double det = GaussJordan(a, b);

  // expected: 60
  cout << "Determinant: " << det << endl;

  // expected: -0.233333 0.166667 0.133333 0.0666667
  //           0.166667 0.166667 0.333333 -0.333333
  //           0.233333 0.833333 -0.133333 -0.0666667
  //           0.05 -0.75 -0.1 0.2
  cout << "Inverse: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }

  // expected: 1.63333 1.3
  //           -0.166667 0.5
```

```
  //          2.36667 1.7
  //          -1.85 -1.35
  cout << "Solution: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
      cout << b[i][j] << ' ';
    cout << endl;
  }
}
```

6.7 Gauss Xor

```
const int MAXN = (1 << 20);
const int MAXLOG = 64;

struct basis
{
    int64_t base[MAXLOG];

    void clear()
    {
            for(int i = MAXLOG - 1; i >= 0; i--)
                    base[i] = 0;
    }

    void add(int64_t val)
    {
        for(int i = MAXLOG - 1; i >= 0; i--)
                    if((val >> i) & 1)
                    {
                            if(!base[i]) { base[i] = val; return; }
                            else val ^= base[i];
                    }
        }

    inline int size()
    {
        int sz = 0;
        for(int i = 0; i < MAXLOG; i++)
                    sz += (bool)(base[i]);
            return sz;
    }

    int64_t max_xor()
    {
            int64_t res = 0;
            for(int i = MAXLOG - 1; i >= 0; i--)
                    if(!((res >> i) & 1) && base[i])
                            res ^= base[i];

            return res;
        }

    bool can_create(int64_t val)
    {
            for(int i = MAXLOG - 1; i >= 0; i--)
                    if(((val >> i) & 1) && base[i])
                            val ^= base[i];

            return (val == 0);
    }
};
```

6.8 Gaussian 1

```
void gauss(vector< vector<double> > &A) {

    int n = A.size();

        for(int i = 0; i < n; i++){
        int r = i;
        for(int j = i+1; j < n; j++)
            if(fabs(A[j][i]) > fabs(A[r][i]))
                r = j;
        if(fabs(A[r][i]) < EPS) continue;
        if(r != i)
            for(int j = 0; j <= n; j++)
                swap(A[r][j], A[i][j]);
        for(int k = 0; k < n; k++){
            if(k != i){
                for(int j = n; j >= i; j--)
                    A[k][j] -= A[k][i]/A[i][i]*A[i][j];
            }
        }
    }
```

```
    // solve: A[x][n]/A[x][x] for each x
}
```

6.9 Gaussian 2

```
const double eps = 1e-9;

// *****may return empty vector

vector<double> gauss(vector<vector<double>> &a)
{
        int n = a.size(), m = a[0].size() - 1;

        vector<int> where(m, -1);
        for(int col = 0, row = 0; col < m && row < n; col++)
    {
        int sel = row;
        for(int i = row; i < n; i++)
                if(abs(a[i][col]) > abs(a[sel][col]))
                        sel = i;

                if(abs(a[sel][col]) < eps) { where[col] = -1; continue; }

        for(int i = col; i <= m; i++)
                        swap(a[sel][i], a[row][i]);
                where[col] = row;

                for(int i = 0; i < n; i++)
                        if(i != row)
                        {
                                if(abs(a[i][col]) < eps) continue;
                double c = a[i][col] / a[row][col];
                for(int j = 0; j <= m; j++)
                    a[i][j] -= c * a[row][j];
                        }

                row++;
    }

    vector<double> ans(m, 0);
    for(int i = 0; i < m; i++)
        if(where[i] != -1)
                        ans[i] = a[where[i]][m] / a[where[i]][i];

        // Validity check?
        // May need to remove the following code

    for(int i = 0; i < n; i++)
        {
                double sum = a[i][m];
                for(int j = 0; j < m; j++)
                        sum -= ans[j] * a[i][j];

                if(abs(sum) > eps) return vector<double>();
        }

        return ans;
}
```

7 Miscellaneous

7.1 Header

```
#pragma comment(linker, "/stack:200000000")
#pragma GCC optimize("unroll-loops")

#include <bits/stdc++.h>
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/tree_policy.hpp>
// #include <ext/pb_ds/detail/standard_policies.hpp>

using namespace std;
// using namespace __gnu_pbds;
// using namespace __gnu_cxx;

typedef long long ll;
typedef unsigned int ul;
typedef unsigned long long ull;
typedef vector <int> vi;
typedef map<int, vector<int> > mvii;
typedef map<int, int> mii;
typedef queue <int> qi;
typedef vector <string> vs;
typedef pair <int, int> pii;
typedef vector<pii > vpii;
```

```
// Order Statistic Tree

/* Special functions:

        find_by_order(k) --> returns iterator to the kth largest
            element counting from 0
        order_of_key(val) --> returns the number of items in a set
            that are strictly smaller than our item
*/

// typedef tree<
// int,
// null_type,
// less<int>,
// rb_tree_tag,
// tree_order_statistics_node_update>
// ordered_set;

#define MP make_pair
#define SORT(a) sort (a.begin(), a.end())
#define REVERSE(a) reverse (a.begin(), a.end())
#define ALL(a) a.begin(), a.end()
#define PI acos(-1)
#define ms(x,y) memset (x, y, sizeof (x))
#define inf 1e9
#define INF 1e16
#define pb push_back
#define MAX 100005
#define debug(a,b) cout<<a<<": "<<b<<endl
#define Debug cout<<"Reached here"<<endl
#define prnt(a) cout<<a<<"\n"
#define mod 1000000007LL
#define FOR(i,a,b) for (int i=(a); i<(b); i++)
#define FORr(i,a,b) for (int i=(a); i>=(b); i--)
#define itrALL(c,itr) for(__typeof((c).begin())
    itr=(c).begin();itr!=(c).end();itr++)
#define lc ((node)<<1)
#define rc ((node)<<1|1)
#define VecPrnt(v) FOR(J,0,v.size()) cout<<v[J]<<" "; cout<<endl
#define endl "\n"
#define PrintPair(x) cout<<x.first<<" "<<x.second<<endl
#define EPS 1e-9
#define ArrPrint(a,st,en) for(int J=st; J<=en; J++) cout<<a[J]<<" ";
    cout<<endl;
```

```
/* Direction Array */

// int fx[]={1,-1,0,0};
// int fy[]={0,0,1,-1};
// int fx[]={0,0,1,-1,-1,1,-1,1};
// int fy[]={-1,1,0,0,1,1,-1,-1};

template <class T> inline T bigmod(T p,T e,T M)
{
    ll ret = 1;
    for(; e > 0; e >>= 1)
    {
        if(e & 1) ret = (ret * p) % M;
        p = (p * p) % M;
    } return (T)ret;
}

template <class T> inline T gcd(T a,T b){if(b==0)return a;return
    gcd(b,a%b);}
template <class T> inline T modinverse(T a,T M){return bigmod(a,M-2,M);}
template <class T> inline T lcm(T a,T b) {a=abs(a);b=abs(b); return
    (a/gcd(a,b))*b;}
template <class T, class X> inline bool getbit(T a, X i) { T t=1; return
    ((a&(t<<i))>0);}
template <class T, class X> inline T setbit(T a, X i) { T t=1;return
    (a|(t<<i)); }
template <class T, class X> inline T resetbit(T a, X i) { T t=1;return
    (a&(~(t<<i)));}

inline ll getnum()
{
    char c = getchar();
    ll num,sign=1;
    for(;c<'0'||c>'9';c=getchar())if(c=='-')sign=-1;
    for(num=0;c>='0'&&c<='9';)
    {
        c-='0';
        num = num*10+c;
        c=getchar();
    }
    return num*sign;
}

inline ll power(ll a, ll b)
```

```
{
        ll multiply=1;
        FOR(i,0,b)
        {
                multiply*=a;
        }
        return multiply;
}

/****** END OF HEADER ******/


int main()
{
    // ios_base::sync_with_stdio(0);
    // cin.tie(NULL); cout.tie(NULL);
    // freopen("in.txt","r",stdin);

    int test, cases=1;



    return 0;
}
```

7.2 Russian Peasant Multiplication

```
// calculate (a*b)%m
// Particularly useful when a, b, m all are large like 1e18
ll RussianPeasantMultiplication(ll a, ll b, ll m)
{
        ll ret=0;

        while(b)
        {
                if(b&1)
                {
                        ret+=a;
                        if(ret>=m) ret-=m;
                }

                a=(a<<1);
                if(a>=m) a-=m;

                b>>=1;
        }

        return ret;
}
```

8 String

8.1 Aho Corasick 2

```
int n;
string s, p[MAX];
map<char, int> node[MAX];
int root, nnode, link[MAX], occ[MAX];
vi ending[MAX], exist[MAX];
// exist[i] has all the ending occurrences of the input strings
void insertword(int idx)
{
        int len = p[idx].size();
        int now = root;
        FOR(i, 0, len)
        {
                if (!node[now][p[idx][i]])
                {
                        node[now][p[idx][i]] = ++nnode;
                        node[nnode].clear();
                }
                now = node[now][p[idx][i]];
        }
// which strings end in node number 'now'?
        ending[now].pb(idx);
}
void populate(int curr)
{
// Because 'suffix-links'. It links a node with the longest proper suffix
        for (auto it : ending[link[curr]])
                ending[curr].pb(it);
}
void populate(vi &curr, int idx)
{
```

```
// So word number it ends in idx-th character of the text
        for (auto it : curr)
        {
                exist[it].pb(idx);
        }
}
void push_links()
{
        queue<int>q;
        link[0] = -1;
        q.push(0);
        while (!q.empty())
        {
                int u = q.front();
                q.pop();
                itrALL(node[u], it)
                {
                        char ch = it->first;
                        int v = it->second;
                        int j = link[u];
                        while (j != -1 && !node[j][ch])
                                j = link[j];
                        if (j != -1)link[v] = node[j][ch];
                        else link[v] = 0;
                        q.push(v);
                        populate(v);
                }
        }
}

void traverse()
{
        int len = s.size();
        int now = root;
        FOR(i, 0, len)
        {
                while (now != -1 && !node[now][s[i]])
                        now = link[now];
                if (now != -1) now = node[now][s[i]];
                else now = 0;
                populate(ending[now], i);
        }
}
```

8.2 Aho Corasick Occurrence Relation

```
// Suppose we have n<=1000 strings. Total summation of the length of
    these strings
// can be 1e7. Now we are given queries. In each query, we are given
    indices of
// two strings and asked if one of them occurs in another as a substring.
// We need to find this relation efficiently. We will use Aho-Corasick.

// The solution is to build a graph where vertices denote indices of
    strings and an edge
// from u to v denotes that string[u] occurs in string[v].

#define ALPHABET_SIZE 26
#define MAX_NODE 1e6
int n; // number of strings
string in[N], p;

int node[MAX_NODE][ALPHABET_SIZE];
int root, nnode, link[MAX_NODE], termlink[MAX_NODE], terminal[MAX_NODE];
bool graph[N][N];

// termlink[u] = a link from node u to a node which is a terminal node
// terminal node is a node where an ending of an input string occurs
// terminal[node] = the index of the string which ends in node

/* Solution:
// For every node of the Aho-Corasick structure find and remember the
nearest terminal node (termlink[u]) in the suffix-link path; Once again
    traverse
all strings through Aho-Corasick. Every time new symbol is added, add an
    arc from the node
corresponding to the current string (in the graph we build, not
    Aho-Corasick) to
the node of the graph corresponding to the nearest terminal in the
    suffix-link path;
The previous step will build all essential arcs plus
some other arcs, but they do not affect the next step in any way;
Find the transitive closure of the graph.
*/

void init()
{
    root=0;
    nnode=0;
```

```
    ms(terminal,-1);
    ms(termlink,-1);
}

void insertword(int idx)
{
        p=in[idx];
    int len=p.size();
    int now=root;

    FOR(i,0,len)
    {
        int x=p[i]-'a';

        if(!node[now][x])
        {
            node[now][x]=++nnode;
        }

        now=node[now][x];
    }

    terminal[now]=idx; // string with index idx ends in now
}

void push_links()
{
    queue<int>q;
    link[0]=-1;
    q.push(0);

    while(!q.empty())
    {
        int u=q.front();
        q.pop();

        for(int i=0; i<ALPHABET_SIZE; i++)
        {
                if(!node[u][i]) continue;

            int v=node[u][i];
            int j=link[u];

            while(j!=-1 && !node[j][i])
                j=link[j];
            if(j!=-1) link[v]=node[j][i];
            else link[v]=0;

                        // Finding nearest terminal nodes
            if(terminal[link[v]]!=-1)
                termlink[v]=link[v];
            else termlink[v]=termlink[link[v]];

            q.push(v);
        }
    }
}

void buildgraph()
{
        FOR(i,0,n)
        {
                int curr=root;

                FOR(j,0,in[i].size())
                {
                        char ch=in[i][j];
                        curr=node[curr][(int)ch-'a'];

                        int st=curr;
                        if(terminal[st]==-1) st=termlink[st];

                        for(int k=st; k>=0; k=termlink[k])
                        {
                                if(terminal[k]==i) continue;
                                if(graph[i][terminal[k]]) break;
                                graph[i][terminal[k]]=true;

                                // cout<<"edge: "<<i<<" "<<terminal[k]<<endl;
                        }
                }
        }
}

// Finally, find transitive closure of the graph. If O(n^3) is possible,
    we can use
// Floyd-Warshall. Otherwise, run dfs from each node and add an edge from
    current starting
```

```
// node to each reachable node. An edge in this transitive closure
    denotes the occurrence relation.
```

8.3 Aho Corasick

```
int n; // n is the number of dictionary word
string s,p; // dictionary words are inputted in p, s is the traversed text

#define MAX_NODE 250004

map<char,int> node[MAX_NODE];
int root, nnode, link[MAX_NODE], endof[504], travis[MAX_NODE];
pii level[MAX_NODE];

void init()
{
    root=0;
    nnode=0;
    travis[root]=0; // number of time a node is traversed by s
    level[root]=MP(0,root); // level, node
    node[root].clear();
}

void insertword(int idx)
{
    int len=p.size();
    int now=root;

    FOR(i,0,len)
    {
        if(!node[now][p[i]])
        {
            node[now][p[i]]=++nnode;
            node[nnode].clear();

            travis[nnode]=0;
            level[nnode]=MP(level[now].first+1,nnode);
        }

        now=node[now][p[i]];
    }

    endof[idx]=now; // end of dictionary word idx
}

void push_links()
{
    queue<int>q;
    link[0]=-1;
    q.push(0);

    while(!q.empty())
    {
        int u=q.front();
        q.pop();

        itrALL(node[u],it)
        {
            char ch=it->first;
            int v=it->second;
            int j=link[u];

            while(j!=-1 && !node[j][ch])
                j=link[j];

            if(j!=-1) link[v]=node[j][ch];
            else link[v]=0;

            q.push(v);
        }
    }
}

void traverse()
{
    int len=s.size();
    int now=root;

    travis[root]++;

    FOR(i,0,len)
    {
        while(now!=-1 && !node[now][s[i]])
            now=link[now];

        if(now!=-1) now=node[now][s[i]];
        else now=0;
```

```
        travis[now]++;
    }

    sort(level,level+nnode+1,greater<pii>());

    FOR(i,0,nnode+1)
    {
        now=level[i].second;
        travis[link[now]]+=travis[now];
    }
}

void driver()
{
    init();
    FOR(i,0,n)
    {
        // input p
        insertword(i);
    }
    // input s
    push_links();
    traverse();
    // number of occurence of word i in s is travis[endof[i]]
}
```

8.4 KMP 2

```
char text[MAX], patt[MAX];
int pi[MAX], n, m;

void Process()
{
        int now=-1;
        pi[0]=-1;

        for(int i=1; i<m; i++)
        {
                while(now!=-1 && patt[now+1]!=patt[i])
                        now=pi[now];
                if(patt[now+1]==patt[i]) pi[i]=++now;
                else pi[i]=now=-1;
        }
}

void Search()
{
        int now=-1;

        for(int i=0; i<n; i++)
        {
                while(now!=-1 && patt[now+1]!=text[i])
                        now=pi[now];
                if(patt[now+1]==text[i]) ++now;
                else now=-1;
                if(now==m-1)
                {
                        cout<<"match at "<<i-now<<endl;
                        now=pi[now]; // match again
                }
        }
}

int main()
{
        // ios_base::sync_with_stdio(0);
        // cin.tie(NULL); cout.tie(NULL);
        // freopen("in.txt","r",stdin);

        cin>>text>>patt;

        n=strlen(text); m=strlen(patt);

        Process();
        Search();

        // FOR(i,0,m) cout<<pi[i]<<" "; cout<<endl;

        return 0;
}
```

8.5 KMP 3

```
string p, t;
int pi[MAX], cnt[MAX];
```

```
void prefixFun()
{
	int now;
	pi[0]=now=-1;

	for(int i=1; i<p.size(); i++)
	{
		while(now!=-1 && p[now+1]!=p[i])
			now=pi[now];

		if(p[now+1]==p[i]) pi[i]=++now;
		else pi[i]=now=-1;
	}
}

int kmpMatch()
{
	int now=-1;
	FOR(i,0,t.size())
	{
		cout<<"now: "<<i<<" "<<now<<endl;
		while(now!=-1 && p[now+1]!=t[i])
			now=pi[now];
		if(p[now+1]==t[i])
		{
			++now;
			cnt[now]++;
		}
		else now=-1;
		if(now+1==p.size())
		{
			// match found
			// cout<<"match and setting "<<now<<" to
			   "<<pi[now]<<endl;
			now=pi[now]; // match again
		}
	}
}

int main()
{
	// ios_base::sync_with_stdio(0);
	// cin.tie(NULL); cout.tie(NULL);
	// freopen("in.txt","r",stdin);
	cin>>t>>p;

	prefixFun();
	FOR(i,0,p.size()) cout<<pi[i]<<" "; cout<<endl;
	prnt(kmpMatch());
	FOR(i,0,p.size()) cout<<cnt[i]<<" "; cout<<endl;
	FORr(i,p.size()-1,0)
	{
		if(pi[i]==-1) continue;
		cnt[pi[i]]+=cnt[i];
	}
	FOR(i,0,p.size()) cout<<cnt[i]<<" "; cout<<endl;

	return 0;
}
```

8.6 Palindromic Tree

```
class PalindromicTree
{
public:
	int s[MAX], Link[MAX], Len[MAX], Edge[MAX][26];
	int node, lastPal, n;
	ll cnt[MAX];

	void init()
	{
		s[n++]=-1;
		Link[0]=1; Len[0]=0;
		Link[1]=1; Len[1]=-1;
		node=2;
	}

	int getLink(int v)
	{
		while(s[n-Len[v]-2]!=s[n-1]) v=Link[v];
		return v;
	}

	void addLetter(int c)
	{
		// cout<<char(c+'a')<<" "<<n<<endl;
```

```
		s[n++]=c;
		lastPal=getLink(lastPal);

		if(!Edge[lastPal][c])
		{
			Len[node]=Len[lastPal]+2;
			Link[node]=Edge[getLink(Link[lastPal])][c];
			cnt[node]++;
			Edge[lastPal][c]=node++;
		}
		else
		{
			cnt[Edge[lastPal][c]]++;
		}

		lastPal=Edge[lastPal][c];
	}

	void clear()
	{
		FOR(i,0,node+1)
		{
			cnt[i]=0;
			ms(Edge[i],0);
		}
		n=0;
		lastPal=0;
	}
} PTA;
```

8.7 String Split by Delimiter

```
template<typename Out>
void split(const std::string &s, char delim, Out result) {
    std::stringstream ss(s);
    std::string item;
    while (std::getline(ss, item, delim)) {
        *(result++) = item;
    }
}

std::vector<std::string> split(const std::string &s, char delim) {
    std::vector<std::string> elems;
    split(s, delim, std::back_inserter(elems));
    return elems;
}
```

8.8 Suffix Array 2

```
// You are given two strings A and B, consisting only of lowercase
    letters from the English alphabet.
// Count the number of distinct strings S, which are substrings of A, but
    not substrings of B

LL substr_count(int n,char *s)
{
	VI cnt(128);
	for(int i=0;i<n;i++)
		cnt[s[i]]++;
	for(int i=1;i<128;i++)
		cnt[i]+=cnt[i-1];
	VI p(n);
	for(int i=0;i<n;i++)
		p[--cnt[s[i]]]=i;
	VVI c(1,VI(n));
	int w=0;
	for(int i=0;i<n;i++)
	{
		if(i==0 || s[p[i]]!=s[p[i-1]]) w++;
		c[0][p[i]] = w-1;
	}

	for(int k=0,h=1;h<n;k++,h*=2)
	{
		VI pn(n);
		for(int i=0;i<n;i++) {
			pn[i] = p[i] - h;
			if(pn[i]<0) pn[i] += n;
		}
		VI cnt(w,0);
		for(int i=0;i<n;i++)
			cnt[c[k][pn[i]]]++;
		for(int i=1;i<w;i++)
			cnt[i]+=cnt[i-1];
		for(int i=n;i--;)
			p[--cnt[c[k][pn[i]]]]=pn[i];
```

```
        w=0;
        c.push_back(VI(n));
        for(int i=0;i<n;i++)
        {
            if(i==0 || c[k][p[i]] != c[k][p[i-1]]) {
                w++;
            } else {
                int i1 = p[i]  + h; if(i1>=n) i1-=n;
                int i2 = p[i-1] + h; if(i2>=n) i2-=n;
                if(c[k][i1]!=c[k][i2]) w++;
            }
            c[k+1][p[i]] = w-1;
        }
    }

    LL ans = LL(n)*(n-1)/2;
    for(int k=1;k<n;k++)
    {
        int i=p[k];
        int j=p[k-1];
        int cur = 0;
        for (int h=c.size(); h--;)
            if (c[h][i] == c[h][j]) {
                cur += 1<<h;
                i += 1<<h;
                j += 1<<h;
            }
        ans-=cur;
    }
    return ans;
}

char s[200005];
int n, m;

void input()
{
    scanf("%s", s);

    n=strlen(s)+1;
    s[n-1]='a'-1;

    scanf("%s", s+n);
    m=strlen(s+n)+1;
    s[n+m-1]='a'-2;
    s[n+m]=0;
}

void solve()
{
    ll p=substr_count(n,s);
    ll r=substr_count(m,s+n);
    ll q=substr_count(n+m,s)-(ll)n*m;

    // cout<<p<<" "<<r<<" "<<q<<endl;

    ll t=q-p-r;

    t=abs(t);

    prnt(p-t);
}

int main()
{
  // ios_base::sync_with_stdio(0);
  // cin.tie(NULL); cout.tie(NULL);
  // freopen("in.txt","r",stdin);

  int test, cases=1;

  input();
  solve();

  return 0;
}
```

8.9 Suffix Array

```
#include <bits/stdc++.h>

using namespace std;

// sa[i] -> ith smallest suffix of the string (indexed from 0)
// height[i] -> Longest common substring between Suffix(i) and
    Suffix(i-1), indexed
// from i=2.
```

```cpp
// rak[i] -> The position of i th index of the main string in height
    array.
// 0 or 1 based ranking (check)

const int N = 2e6+5;
int wa[N],wb[N],wv[N],wc[N];
int r[N],sa[N],rak[N], height[N];

int cmp(int *r,int a,int b,int l)
{
     return r[a] == r[b] && r[a+l] == r[b+l];
}

void da(int *r,int *sa,int n,int m)
{
     int i,j,p,*x=wa,*y=wb,*t;
     for( i=0;i<m;i++) wc[i]=0;
     for( i=0;i<n;i++) wc[x[i]=r[i]] ++;
     for( i=1;i<m;i++) wc[i] += wc[i-1];
     for( i= n-1;i>=0;i--)sa[--wc[x[i]]] = i;
     for( j= 1,p=1;p<n;j*=2,m=p){
          for(p=0,i=n-j;i<n;i++)y[p++] = i;
          for(i=0;i<n;i++)if(sa[i] >= j) y[p++] = sa[i] - j;
          for(i=0;i<n;i++)wv[i] = x[y[i]];
          for(i=0;i<m;i++) wc[i] = 0;
          for(i=0;i<n;i++) wc[wv[i]] ++;
          for(i=1;i<m;i++) wc[i] += wc[i-1];
          for(i=n-1;i>=0;i--) sa[--wc[wv[i]]] = y[i];
          for(t=x,x=y,y=t,p=1,x[sa[0]] = 0,i=1;i<n;i++) x[sa[i]]=
              cmp(y,sa[i-1],sa[i],j) ? p-1:p++;

     }
}

void calheight(int *r,int *sa,int n)
{

     int i,j,k=0;
     for(i=1;i<=n;i++) rak[sa[i]] = i;
     for(i=0;i<n;height[rak[i++]] = k ) {

          for(k?k--:0, j=sa[rak[i]-1] ; r[i+k] == r[j+k] ; k ++) ;
     }

}
```

```cpp
int dp[N][22];

void initRMQ(int n)
{
     for(int i= 1;i<=n;i++) dp[i][0] = height[i];
     for(int j= 1; (1<<j) <= n; j ++ ){
          for(int i = 1; i + (1<<j) - 1 <= n ; i ++ ) {
               dp[i][j] = min(dp[i][j-1] , dp[i + (1<<(j-1))][j-1]);
          }
     }

}
int askRMQ(int L,int R)
{
     int k = 0;
     while((1<<(k+1)) <= R-L+1) k++;
     return min(dp[L][k], dp[R - (1<<k) + 1][k]);
}

int main()
{

       return 0;
}
```

8.10 Suffix Automata

```cpp
// Counts number of distinct substrings

struct suffix_automaton
{
   map<char, int> to[MAX];
   int len[MAX], link[MAX];
   int last, psz = 0;

   void add_letter(char c)
   {
      int p = last, cl, q;
      if(to[p].count(c))
      {
         q = to[p][c];
         if(len[q] == len[p] + 1)
```

```
        {
            last = q;
            return;
        }

        cl = psz++;
        len[cl] = len[p] + 1;
        to[cl] = to[q];
        link[cl] = link[q];
        link[q] = cl;
        last = cl;

        for(; to[p][c] == q; p = link[p])
            to[p][c] = cl;

        return;
    }

    last = psz++;
    len[last] = len[p] + 1;

    for(; to[p][c] == 0; p = link[p])
        to[p][c] = last;

    if(to[p][c] == last)
    {
        link[last] = p;
        return;
    }

    q = to[p][c];
    if(len[q] == len[p] + 1)
    {
        link[last] = q;
        return;
    }

    cl = psz++;
    len[cl] = len[p] + 1;
    to[cl] = to[q];
    link[cl] = link[q];
    link[q] = cl;
    link[last] = cl;

    for(; to[p][c] == q; p = link[p])
            to[p][c] = cl;
    }

    void clear()
    {
        for(int i = 0; i < psz; i++)
            len[i] = 0, link[i] = 0, to[i].clear();
        psz = 1;
        last = 0;
    }

    void init(string s)
    {
        clear();
        for(int i = 0; i < s.size(); i++)
            add_letter(s[i]);
    }

    suffix_automaton() {psz = 0; clear();}
};

string s;
suffix_automaton SA;
ll cnt[MAX];
vi endpos[MAX];

int main()
{
    // ios_base::sync_with_stdio(0);
    // cin.tie(NULL); cout.tie(NULL);
    // freopen("in.txt","r",stdin);

    int test, cases=1;

    cin>>s;

    SA.clear();
    FOR(i,0,s.size())
        SA.add_letter(s[i]), cnt[SA.last]++;

    FOR(i,0,SA.psz)
    {
        endpos[SA.len[i]].pb(i);
    }
```

```
    ll ans=0;

    FORr(i,SA.psz-1,1)
    {
        for(auto it: endpos[i])
        {
                cnt[SA.link[it]]+=cnt[it];
                ans+=(SA.len[it]-SA.len[SA.link[it]]); // distinct
                     occurrences
            // cnt[it] has occurrence of substring ending at node it
        }
    }

    // cnt[x] has occurrences of state x
    // To calculate occurrence of an input string, we visit the automata
         using the letters
    // of the input string and find the last_state where it finishes
    // The cnt[last_state] should be the occurrence of this string

    prnt(ans);

    return 0;
}
```

8.11 Trie 1

```
struct Node
{
        int cntL, cntR, lIdx, rIdx;
        Node()
        {
                cntL = cntR = 0;
                lIdx = rIdx = -1;
        }
};
Node Tree[MAX];
int globalIdx = 0;
class Trie
{
public:
        void insert(int val, int idx, int depth)
        {
                for (int i = depth - 1; i >= 0; i--)
                {
                        bool bit = val & (1 << i);
                        // cout<<"bit now: "<<bit<<endl;
                        if (bit)
                        {
                                Tree[idx].cntR++;
                                if (Tree[idx].rIdx == -1)
                                {
                                        Tree[idx].rIdx = ++globalIdx;
                                        idx = globalIdx;
                                }
                                else idx = Tree[idx].rIdx;
                        }
                        else
                        {
                                Tree[idx].cntL++;
                                if (Tree[idx].lIdx == -1)
                                {
                                        Tree[idx].lIdx = ++globalIdx;
                                        idx = globalIdx;
                                }
                                else idx = Tree[idx].lIdx;
                        }
                }
        }
        int query(int val, int compVal, int idx, int depth)
        {
                int ans = 0;
                for (int i = depth - 1; i >= 0; i--)
                {
                        bool valBit = val & (1 << i);
                        bool compBit = compVal & (1 << i);
                        if (compBit)
                        {
                                if (valBit)
                                {
                                        ans += Tree[idx].cntR;
                                        idx = Tree[idx].lIdx;
                                }
                                else
                                {
                                        ans += Tree[idx].cntL;
                                        idx = Tree[idx].rIdx;
                                }
                        }
```

```
                else
                {
                        if (valBit)
                        {
                                idx = Tree[idx].rIdx;
                        }
                        else
                        {
                                idx = Tree[idx].lIdx;
                        }
                }
                if (idx == -1) break;
        }
        return ans;
    }
    void clear()
    {
        for (int i = 0; i <= globalIdx; i++)
        {
                Tree[i].cntL = 0;
                Tree[i].cntR = 0;
                Tree[i].rIdx = -1;
                Tree[i].lIdx = -1;
        }
        globalIdx = 0;
    }
};
int main()
{
    // ios_base::sync_with_stdio(0);
    // cin.tie(NULL); cout.tie(NULL);
    // freopen("in.txt","r",stdin);
    // Given an array of positive integers you have to print the
       number of
    // subarrays whose XOR is less than K.
    int test, n, k, x;
    Trie T;
    scanf("%d", &test);
    while (test--)
    {
        scanf("%d%d", &n, &k);
        T.insert(0, 0, 20);
        int pre = 0;
        ll ans = 0;
        FOR(i, 0, n)
        {
                scanf("%d", &x);
                pre ^= x;
                // prnt(pre);
                ans += T.query(pre, k, 0, 20);
                T.insert(pre, 0, 20);
        }
        printf("%lld\n", ans);
        T.clear();
    }
    return 0;
}
```

8.12 Trie 2

```
const int MaxN = 100005;
int sz;

int nxt[MaxN][55];
int en[MaxN];

bool isSmall(char ch)
{
    return ch>='a' && ch<='z';
}

int getId(char ch)
{
    if(isSmall(ch)) return ch-'a';
    else return ch-'A'+26;
}

void insert (char *s, int l)
{
    int v = 0;

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

        int c=getId(s[i]);

        if (nxt[v][c]==-1)
        {
            ms(nxt[sz],-1);
```

```
                nxt[v][c]=sz++;
                en[sz]=0;
                // created[sz] = true;
            }

            v = nxt[v][c];
        }
        ++en[v];
}

int search (char *tmp, int l) {

        int v = 0;

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

            int c=getId(tmp[i]);

            if (nxt[v][c]==-1)
                return 0;

            v = nxt[v][c];
        }
        return en[v];
}

void init()
{
        sz=1;
        en[0]=0;
        ms(nxt[0],-1);
}
```