

# Codebook- Team Know\_no\_algo

Ankesh Gupta, Ronak Agarwal, Anant Chhajwani

## Contents

<b>1</b>	<b>Format</b>	<b>2</b>			
1.1	Format c++ . . . . .	2			
1.2	Format Java . . . . .	2			
<b>2</b>	<b>Strings</b>	<b>3</b>			
2.1	KMP . . . . .	3			
2.2	Manacher . . . . .	3			
2.3	Suffix_Array . . . . .	3			
2.4	Z algo . . . . .	4			
2.5	Hashing . . . . .	4			
2.6	Trie . . . . .	4			
<b>3</b>	<b>Trees</b>	<b>5</b>			
3.1	Centroid Tree . . . . .	5			
3.2	Heavy Light Decomposition . . . . .	5			
3.3	Heavy Light Trick . . . . .	6			
3.4	LCA . . . . .	7			
3.5	LCA Tree . . . . .	7			
<b>4</b>	<b>Graph and Matching, Flows</b>	<b>7</b>			
4.1	AP and Bridges . . . . .	7			
4.2	Euler Walk . . . . .	8			
4.3	Bipartite Matching . . . . .	8			
4.4	Dinic- Maximum Flow $O(EV^2)$ . . . . .	9			
4.5	Minimum Cost Bipartite Matching $O(V^3)$ . . . . .	10			
4.6	Minimum Cost Maximum Flow . . . . .	11			
4.7	General Unweighted Maximum Matching (Edmonds' algorithm) . . . . .	12			
			4.8	König's Theorem (Text) . . . . .	13
			4.9	Minimum Edge Cover (Text) . . . . .	13
<b>5</b>	<b>Data Structures</b>	<b>14</b>			
5.1	Persistent Segment Tree . . . . .	14			
5.2	BIT- Point Update + Range Sum . . . . .	14			
5.3	BIT- Range Update + Range Sum . . . . .	15			
5.4	BIT- 2D . . . . .	15			
<b>6</b>	<b>Math</b>	<b>16</b>			
6.1	Convex Hull . . . . .	16			
6.2	FFT . . . . .	16			
6.3	Convex Hull Trick . . . . .	17			
6.4	Miscellaneous Geometry . . . . .	17			
6.5	Gaussian elimination for square matrices of full rank; finds inverses and determinants . . . . .	20			
<b>7</b>	<b>Number Theory Reference</b>	<b>20</b>			
7.1	Fast factorization (Pollard rho) and primality testing (Rabin–Miller) . . . . .	20			
7.2	Modular arithmetic and linear Diophantine solver . . . . .	22			
7.3	Polynomial Coefficients (Text) . . . . .	23			
7.4	Möbius Function (Text) . . . . .	23			
7.5	Burnside's Lemma (Text) . . . . .	23			
<b>8</b>	<b>Miscellaneous</b>	<b>23</b>			
8.1	2-SAT . . . . .	23			
8.2	Stable Marriage Problem (Gale–Shapley algorithm) . . . . .	24			
<b>9</b>	<b>Credits</b>	<b>24</b>			

# 1 Format

## 1.1 Format c++

```
#include <bits/stdc++.h> #define long long long ←
int
using namespace std; #define Max 100010
#define mp make_pair #define pb push_back
#define INF 1e16 #define INF2 1e9+9
#define pi 3.141592653589 #define x first
#define y second
long cons;
long check(long a){
    if(a>=cons)a%=cons;
    return a;
}
long check2(long a){
    a%=cons ;
    if(a<0) a+=cons ;
    return a;
}
long GCD(long a,long b){
    if(b==0)
        return a;
    return GCD(b,a%b);
}
long exp(long a,long n){
    if(n==0) return 1;
    if (n==1) return check(a);
    long b=exp(a,n/2);
    if(n%2==0) return check(b*b);
    return check(b*check(b*a));
}
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    cons=1000000007 ;
}
```

## 1.2 Format Java

```
import java.io.* ;
import java.util.* ;
import java.math.* ;
import java.text.* ;
import static java.lang.Math.min ;
```

```
import static java.lang.Math.max ;
public class Main{
    public static void main(String args[]) throws ←
        IOException{
        Solver s = new Solver() ;
        s.init() ;
        s.Solve() ;
        s.Finish() ;
    }
}
class pair implements Comparable<pair>{
    long x,y ;
    pair(long x,long y){
        this.x = x ; this.y=y ;
    }
    public int compareTo(pair p){
        return (this.x<p.x ? -1 : (this.x>p.x ? 1 : (this←
            .y<p.y ? -1 : (this.y>p.y ? 1 : 0))) ;
    }
}
class Solver{
    void Solve() throws IOException{
    }
    void init(){
        op = new PrintWriter(System.out) ;
        ip = new Reader(System.in) ;
    }
    void Finish(){
        op.flush() ;
        op.close() ;
    }
    void p(Object o){
        op.print(o) ;
    }
    void pln(Object o){
        op.println(o) ;
    }
    PrintWriter op ;
    Reader ip ;
}
class Reader {
    BufferedReader reader;
    StringTokenizer tokenizer;
    Reader(InputStream input) {
        reader = new BufferedReader(
            new InputStreamReader(input) );
        tokenizer = new StringTokenizer(""); ;
    }
    String s() throws IOException {
        while (!tokenizer.hasMoreTokens()){
            tokenizer = new StringTokenizer(
                reader.readLine()) ;
        }
    }
}
```

```

    }
    return tokenizer.nextToken();
}
int i() throws IOException {
    return Integer.parseInt(s());
}
long l() throws IOException{
    return Long.parseLong(s());
}
double d() throws IOException {
    return Double.parseDouble(s());
}
}

```

## 2 Strings

### 2.1 KMP

```

//Takes an array of characters and calculate
//lcp[i] where lcp[i] is the longest proper suffix ←
//of the
//string c[0..i] such that it is also a prefix of ←
//the string.
int[] kmp(char c[],int n){
    int lcp[] = new int[n];
    for(int i=1 ; i<n ; i++){
        int j = lcp[i-1];
        while(j!=0 && c[i]!=c[j]) j = lcp[j-1];
        if(c[i]==c[j]) j++;
        lcp[i]=j;
    }
    return lcp;
}

```

### 2.2 Manacher

```

//Given an array of characters in arr and the ←
//length
//of the array as n it simply finds the longest ←
//palindromic substring

```

```

//at each position with that position as the center ←
//of the palindrome.
// Array is 0-indexed
int[] Manacher(char c[],int n){
    int P[] = new int[n+1];
    int R=0,C=1;
    for(int i=1 ; i<=n ; i++){
        int rad=-1;
        if(i<=R)
            rad = min(P[2*C-i],(R-i));
        else
            rad = 0;
        while((i+rad)<=n && (i-rad)>=1 && c[i-rad]==c[i+rad])
            rad++;
        P[i]=rad-1;
        if((i+rad)>R){
            C = i;
            R = i+rad;
        }
    }
    return P;
}

```

### 2.3 Suffix Array

```

int match(char t[],char s[],int pos,int n){
    for(int i=0 ; i<t.length ; i++){
        if(pos+i==n)
            return 1;
        else if(t[i]!=s[pos+i])
            return (t[i]<s[pos+i] ? -1 : 1);
        return 0;
    }
}
int[] SufTrans(int P[][],int n){
    int suf[] = new int[n];
    for(int i=0 ; i<n ; i++) suf[P[19][i]] = i;
    return suf;
}
int LCP(int i,int j,int P[][],int n){
    if(i==j) return (n-i+1);
    int match=0;
    for(int k=19 ; i<n && j<n && k>=0 ; k--){
        if(P[k][i]==P[k][j]){
            match+=(1<<k);
            i+=(1<<k);
            j+=(1<<k);
        }
    }
}

```

```

    }
    return match ;
}
int [][] suffix_array(char c[],int n){
    class Tuple implements Comparable<Tuple>{
        int idx ; pair p ;
        Tuple(int _idx,pair _p){
            idx = _idx ; p=_p ;
        }
        public int compareTo(Tuple _t){
            return p.compareTo(_t.p) ;
        }
    }
    int P[][] = new int[20][n] ;
    if(n!=1)
        for(int i=0 ; i<n ; i++) P[0][i] = (int) c[i] ;
    else
        P[0][0] = 0 ;
    for(int i=1,pow2=1 ; i<20 ; pow2<=1,i++){
        Tuple L[] = new Tuple[n] ;
        for(int j=0 ; j<n ; j++){
            int y = ((j+pow2)<n ? P[i-1][j+pow2] : -1) ;
            L[j] = new Tuple(j,new pair(P[i-1][j],y)) ;
        }
        Arrays.sort(L) ;
        for(int j=0 ; j<n ; j++)
            if(j>0 && L[j].compareTo(L[j-1])==0)
                P[i][L[j].idx] = P[i][L[j-1].idx] ;
            else
                P[i][L[j].idx] = j ;
    }
    return P ;
}

```

## 2.4 Z algo

```

//Given an array of characters in c and
// length of array is n, find the z-array
//that is z[i]=longest prefix match of suffix
//at i and the original string
int[] Z_algo(char c[],int n){
    int Z[] = new int[n] ;
    int L=0,R=0 ;
    for(int i=1 ; i<n ; i++){
        if(i>R){
            L=i ; R=i ;
            while(R<n && c[R]==c[R-L]) R++ ;
            R-- ; Z[i] = (R-L+1) ;
        }
    }
}

```

```

    }else{
        int j = i-L ;
        if(Z[j]<(R-i+1))
            Z[i]=Z[j] ;
        else{
            L=i ;
            while(R<n && c[R]==c[R-L]) R++ ;
            R-- ; Z[i] = (R-L+1) ;
        }
    }
    return Z ;
}

```

## 2.5 Hashing

```

vector<long> hashed1[10*Max];
vector<long> hashed2[10*Max];
long p1=2350490027,p2=1628175011;
long p3=2911165193,p4=1040332871;
2350490027,2125898167,1628175011,1749241873,
1593209441,1524872353,1040332871,2911165193,
1387346491,2776808933
void calc_hashed(int ind,vector<long> &hashed,long ←
    prime){
    long val=1;
    int x=neighbour[ind].size();
    hashed.resize(x);
    for(int i=0;i<x;i++){
        if(i==0)
            hashed[i]=neighbour[ind][i];
        else
            hashed[i]=check(hashed[i-1]+neighbour[ind][i]*←
                val) ;
        val=check(val*prime);
    }
}

```

## 2.6 Trie

```

struct node{
    int ind;
    node *arr[26] ;
}

```

```

};
node* getnode(int ind){
    node *temp=new node() ;
    temp->ind=ind ;
    for(int i=0;i<26;i++){
        temp->arr[i]=NULL ;
    }
    return temp ;
}
void insert(node *root,string &s,int pos){
    int x=(s.length()) ;
    for(int i=0;i<x;i++){
        int ch=s[i]-97 ;
        if(root->arr[ch]==NULL)
            root->arr[ch]=getnode(pos) ;
        root=root->arr[ch] ;
    }
}

```

```

queue<pair<int,int> > q;
q.push({i,-1});
while(!q.empty()){
    auto itr=q.front();
    q.pop();
    calc_size(itr.x,-1);
    int centroid=getCentroid(itr.x,size[itr.x],-1);
    centroid_parent[centroid]=itr.y;
    for(auto itr2:graph[centroid]){
        if(usable[itr2]){
            q.push({itr2,centroid});
        }
    }
    usable[centroid]=false;
}
}

```

## 3 Trees

### 3.1 Centroid Tree

```

vector<int> graph[3*Max];
int size[3*Max];
bool usable[3*Max];
int centroid_parent[3*Max];
void calc_size(int i,int pa){
    size[i]=1;
    for(auto itr:graph[i]){
        if(itr!=pa && usable[itr]){
            calc_size(itr,i);
            size[i]+=size[itr];
        }
    }
}
int getCentroid(int i,int len,int pa){
    for(auto itr:graph[i]){
        if(itr!=pa && usable[itr]){
            if(size[itr]>(len/2))
                return getCentroid(itr,len,i);
        }
    }
    return i;
}
void build_centroid(int i,int coun){

```

```

int chainNo[Max];
int pos_in_chain[Max];
int parent_in_chain[Max];
int parent[Max];
int chain_count=0;
int total_in_chain[Max];
int pos_count=0;
vector<int> graph[Max];
int arr[Max];
int subtree_count[Max];
int max_in_subtree[Max];
int height[Max];
vector<vector<pair<int,int> > > vec;
int max_elem,max_count;
void simple_dfs(int i){
    subtree_count[i]=1;
    int max_val=0;
    int ind=-1;
    for(auto itr:graph[i]){
        height[itr]=1+height[i];
        simple_dfs(itr);
        subtree_count[i]+=subtree_count[itr];
        if(max_val<subtree_count[itr]){
            max_val=subtree_count[itr];
            ind=itr;
        }
    }
    max_in_subtree[i]=ind;
}

```

### 3.2 Heavy Light Decomposition

```

void dfs(int i){
    if(pos_count==0)
        parent_in_chain[chain_count]=i;
    chainNo[i]=chain_count;
    pos_in_chain[i]=++pos_count;
    total_in_chain[chain_count]++;
    if(max_in_subtree[i]!=-1){
        dfs(max_in_subtree[i]);
    }
    for(auto itr:graph[i]){
        if(itr!=max_in_subtree[i]){
            chain_count++;
            pos_count=0;
            dfs(itr);
        }
    }
}
int pos;int chain;int val;
void update(int s,int e,int n){
    if(pos>e || pos<s)
        return;
    vec[chain][n]={val,1};
    if(s==e)
        return;
    int mid=(s+e)>>1;
    update(s,mid,2*n);
    update(mid+1,e,2*n+1);
    if(vec[chain][2*n].x<vec[chain][2*n+1].x)
        vec[chain][n]=vec[chain][2*n+1];
    else if(vec[chain][2*n].x>vec[chain][2*n+1].x)
        vec[chain][n]=vec[chain][2*n];
    else{
        vec[chain][n]={vec[chain][2*n].x,vec[chain][2*n].x+
            y+vec[chain][2*n+1].y};
    }
}
int qs;int qe;
void query_tree(int s,int e,int n){
    if(s>qe || qs>e)
        return;
    if(s>=qs && e<=qe){
        if(vec[chain][n].x>max_elem){
            max_elem=vec[chain][n].x;
            max_count=vec[chain][n].y;
        }
        else if(vec[chain][n].x==max_elem){
            max_count+=vec[chain][n].y;
        }
        return;
    }
    if(vec[chain][n].x < max_elem)
        return;
    int mid=(s+e)>>1;

```

```

    query_tree(s,mid,2*n);
    query_tree(mid+1,e,2*n+1);
}
void query(int i){
    if(i==-1)
        return;
    qs=1;qe=pos_in_chain[i];chain=chainNo[i];
    query_tree(1,total_in_chain[chainNo[i]],1);
    i=parent[parent_in_chain[chainNo[i]]];
    query(i);
}

```

### 3.3 Heavy Light Trick

```

void dfs(int i,int pa){
    int coun=1 ;
    for(auto itr:a[i]){
        if(itr.x!=pa){
            prod[itr.x]=check(prod[i]*itr.y) ;
            dfs(itr.x,i);
            coun+=siz[itr.x] ;
        }
    }
    siz[i]=coun ;
}
long ans=0 ;
void add(int i,int pa,int x){
    coun[mapped_prod[i]]+=x ;
    for(auto itr:a[i])
        if(itr.x!=pa && !big[itr.x])
            add(itr.x,i,x) ;
}
void solve(int i,int pa){
    long temp=check(multi*inv[i]);
    int xx=m[temp];
    ans+=coun[xx];
    for(auto itr:a[i])
        if(itr.x!=pa && !big[itr.x])
            solve(itr.x,i) ;
}
void dfs2(int i,int pa,bool keep){
    int mx=-1,bigc=-1;
    for(auto itr:a[i]){
        if(itr.x!=pa){
            if(siz[itr.x]>mx)
                mx=siz[itr.x],bigc=itr.x;
        }
    }
}

```

```

for(auto itr:a[i]){
    if(itr.x!=pa && itr.x!=bigc)
        dfs2(itr.x,i,0);
}
if(bigc!=-1){
    dfs2(bigc,i,1);
    big[bigc]=true;
}
multi=check(p*check(prod[i]*prod[i]));
long temp=check(p*prod[i]);
ans+=coun[m[temp]];
coun[mapped_prod[i]]++;
for(auto itr:a[i])
    if(itr.x!=pa && !big[itr.x]){
        solve(itr.x,i);
        add(itr.x,i,1);
    }
if(bigc!=-1)
    big[bigc]=false;
if(keep==0)
    add(i,pa,-1);
}

```

### 3.4 LCA

```

int pa[21][3*Max], level[3*Max];
int lca(int u,int v){
    if(level[u]>level[v])return lca(v,u);
    for(long i=19;i>=0 && level[v]!=level[u];i--){
        if(level[v]>=level[u]+(1<<i))
            v=pa[i][v];
    }
    if(u==v)return u;
    for(long i=19;i>=0;i--){
        if(pa[i][u]!=pa[i][v]){
            u=pa[i][u];v=pa[i][v];
        }
    }
    return pa[0][u];
}

```

### 3.5 LCA Tree

```

vpi auxTree[N];
int parent[N];
ll parWgt[N];
int conAuxTree(set<int, disComp> &nodes) {
    vi originalNodes(nodes.begin(), nodes.end());
    for (int i=0; i<originalNodes.size()-1; i++) {
        nodes.insert(LCA(originalNodes[i], ←
            originalNodes[i+1]));
    }
    int root = *nodes.begin();
    parent[root] = 0;
    int cur = root;
    auto sit = next(nodes.begin());
    while (sit != nodes.end()) {
        while (!isAnc(cur, *sit)) {
            assert(cur);
            cur = parent[cur];
        }
        parent[*sit] = cur;
        parWgt[*sit] = rootDis[*sit] - rootDis[cur]←
        ];
        auxTree[cur].push_back({*sit, parWgt[*sit]←
        });
        cur = *sit;
        ++sit;
    }
    return root;
}

```

## 4 Graph and Matching, Flows

### 4.1 AP and Bridges

```

// Finds bridges and cut vertices
// Receives:
// N: number of vertices
// l: adjacency list
// Gives:
// vis, seen, par (used to find cut vertices)
// ap - 1 if it is a cut vertex, 0 otherwise
// brid - vector of pairs containing the bridges
typedef pair<int, int> PII;
int N;
vector <int> l[MAX];
vector <PII> brid;

```

```

int vis[MAX], seen[MAX], par[MAX], ap[MAX];
int cnt, root;
void dfs(int x){
    if(vis[x] != -1)
        return;
    vis[x] = seen[x] = cnt++;
    int adj = 0;
    for(int i = 0; i < (int)l[x].size(); i++){
        int v = l[x][i];
        if(par[x] == v) continue;
        if(vis[v] == -1){
            adj++;
            par[v] = x;
            dfs(v);
            seen[x] = min(seen[x], seen[v]);
            if(seen[v] >= vis[x] && x != root)
                ap[x] = 1;
            if(seen[v] == vis[v])
                brid.push_back(make_pair(v, x));
        }
        else{
            seen[x] = min(seen[x], vis[v]);
            seen[v] = min(seen[x], seen[v]);
        }
    }
    if(x == root) ap[x] = (adj>1);
}
void bridges(){
    brid.clear();
    for(int i = 0; i < N; i++){
        vis[i] = seen[i] = par[i] = -1;
        ap[i] = 0;
    }
    cnt = 0;
    for(int i = 0; i < N; i++){
        if(vis[i] == -1){
            root = i;
            dfs(i);
        }
    }
}

```

## 4.2 Euler Walk

```

vector<pair<int,int> > graph[202];
bool visited[202];
vector<int> odd;
bool used_edges[41000];
stack<int> s;

```

```

int tot_edges;
int counter[202];
void dfs(int i)
{
    visited[i]=true;

    int len=graph[i].size();
    if(len&1)
        odd.pb(i);

    for(auto itr:graph[i])
        if(!visited[itr.x])
            dfs(itr.x);
}
void euler_tour(int i)
{
    visited[i]=true;
    s.push(i);

    int x=graph[i].size();
    while(counter[i]<x)
    {
        auto itr=graph[i][counter[i]];
        counter[i]++;

        if(!used_edges[itr.y])
        {
            used_edges[itr.y]=true;
            if(itr.y<=tot_edges)
                cout<<i<<" "<<itr.x<<"\n";
            euler_tour(itr.x);
        }
    }
    s.pop();
}

```

## 4.3 Bipartite Matching

```

vector<pair<int,pair<int,bool> > > graph[1000];
vector<bool> edge_use,visited;
vector<int> parent,edge_number;

void dfs(int i){
    visited[i]=true;
    for(auto itr:graph[i]){
        if(visited[itr.x])
            continue;
    }
}

```



```

if(edge_use[itr.y.x] && itr.y.y){
    parent[itr.x]=i;
    edge_number[itr.x]=itr.y.x;
    dfs(itr.x);
}
else if(!edge_use[itr.y.x] && (!itr.y.y)){
    parent[itr.x]=i;
    edge_number[itr.x]=itr.y.x;
    dfs(itr.x);
}
}
}

void edge_reverse(int t){
    if(t==0)
        return;
    edge_use[edge_number[t]]=!edge_use[edge_number[t←
    ]];
    edge_reverse(parent[t]);
}
// |l|,|r| are the number of vertices in the left ←
// side and right side respectively.
int s=0;
int t=(|l|+|r|+1) ;
for(int i=1;i<=h;i++){
    ++edge_count;
    graph[0].pb({i,{edge_count,true}});
    graph[i].pb({0,{edge_count,false}});
}
int matching=0;
edge_use.resize(edge_count+1);
visited.resize(t+1);
edge_number.resize(t+1);
parent.resize(t+1);
for(int i=0;i<=edge_count;i++){
    edge_use[i]=true;
}
while(true){
    for(int i=0;i<=t;i++){
        visited[i]=false;
        edge_number[i]=-1;
        parent[i]=-1;
    }
    dfs(s);
    if(!visited[t])
        break;
    edge_reverse(t);
    matching++;
}

```

## 4.4 Dinic- Maximum Flow $O(EV^2)$

```

const int N = 20005 ;
const int E = N*1005 ;
int t, n, m;
int par[N];
/* START DINIC */
int nodes, edges;
int eu[E], ev[E], ef[E], ec[E];
int dist[N], q[N], ed[N];
vector<int> adj[N];
void init(int n) {
    ::nodes = n;
    ::edges = 0;
    for (int i = 0; i < nodes; ++i)
        adj[i].clear();
}
int newedge(int u, int v, int flow, int cap) {
    eu[edges] = u;
    ev[edges] = v;
    ef[edges] = flow;
    ec[edges] = cap;
    return edges++;
}
void addedge(int u, int v, int cap) {
    int uv = newedge(u, v, 0, cap);
    int vu = newedge(v, u, 0, 0);
    adj[u].push_back(uv);
    adj[v].push_back(vu);
}
bool bfs(int src, int snk) {
    memset(dist, -1, sizeof(int) * nodes);
    int h = 0, t = 0;
    dist[src] = 0;
    q[t++] = src;
    while (h != t && dist[snk] == -1) {
        int u = q[h++]; if (h == N) h = 0;
        for (int e : adj[u]) {
            int v = ev[e];
            if (dist[v] < 0 && ef[e] < ec[e]) {
                dist[v] = dist[u] + 1;
                q[t++] = v;
                if (t == N) t = 0;
            }
        }
    }
    return ~dist[snk];
}
bool dfs(int u, int snk, int flow) {
    if (flow <= 0) return 0;
    if (u == snk) return flow;

```

```

for (int& i = ed[u]; i < (int) adj[u].size(); ++i) {
    int e = adj[u][i];
    int v = ev[e];
    if (dist[u] + 1 == dist[v]) {
        int fl = min(flow, ec[e] - ef[e]);
        int df = dfs(v, snk, fl);
        if (df == 0) continue;
        ef[e] += df;
        ef[e^1] -= df;
        return df;
    }
}
return 0;
}
int dinic(int src, int snk) {
    int mf = 0;
    while (bfs(src, snk)) {
        memset(ed, 0, sizeof(int) * nodes);
        int df;
        while (df = dfs(src, snk, INT_MAX))
            mf += df;
    }
    return mf;
}
int main() {
    scanf("%d", &t);
    while (t--) {
        scanf("%d%d", &n, &m);
        int source = n + m;
        int sink = source + 1;
        init(sink + 1);
        for (int i = 2; i <= n; ++i) {
            scanf("%d", &par[i]);
            addedge(par[i] - 1, i - 1, INT_MAX);
        }
        for (int i = 1; i <= n; ++i) {
            addedge(i - 1, sink, 1);
        }
        for (int i = 0; i < m; ++i) {
            addedge(source, i + n, 1);
            int len;
            scanf("%d", &len);
            for (int j = 0; j < len; ++j) {
                int ai;
                scanf("%d", &ai);
                addedge(i + n, ai - 1, 1);
            }
        }
        printf("%d\n", dinic(source, sink));
    }
}

```

## 4.5 Minimum Cost Bipartite Matching $O(V^3)$

```

// Min cost bipartite matching via shortest ←
// augmenting path
// This is an  $O(n^3)$  implementation of a shortest ←
// augmenting path
// algorithm for finding min cost perfect matchings ←
// in dense
// graphs. In practice, it solves 1000x1000 ←
// problems in around 1 second.
// cost[i][j] = cost for pairing left node i with ←
// right node j
// Lmate[i] = index of right node that left node i ←
// pairs with
// Rmate[j] = index of left node that right node j ←
// pairs with
// The values in cost[i][j] may be positive or ←
// negative. To perform
// maximization, simply negate the cost[][] matrix.
typedef vector<long> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
long MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
    int n = int(cost.size());
    // construct dual feasible solution
    VD u(n); VD v(n);
    for (int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
    }
    for (int j = 0; j < n; j++) {
        v[j] = cost[0][j] - u[0];
        for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
    } // construct primal solution satisfying ←
    // complementary slackness
    Lmate = VI(n, -1);
    Rmate = VI(n, -1);
    int mated = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (Rmate[j] != -1) continue;
            if ((cost[i][j] - u[i] - v[j]) == 0) {
                Lmate[i] = j;
                Rmate[j] = i;
                mated++;
                break;
            }
        }
    }
}

```

```

}
}
VD dist(n); VI dad(n); VI seen(n);
// repeat until primal solution is feasible
while (mated < n) { // find an unmatched left ←
    node
    int s = 0;
    while (Lmate[s] != -1) s++; // initialize ←
    Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
        dist[k] = cost[s][k] - u[s] - v[k];
    int j = 0;
    while (true) { // find closest
        j = -1;
        for (int k = 0; k < n; k++) {
            if (seen[k]) continue;
            if (j == -1 || dist[k] < dist[j]) j = k;
        }
        seen[j] = 1; // termination condition
        if (Rmate[j] == -1) break; // relax ←
        neighbors
        const int i = Rmate[j];
        for (int k = 0; k < n; k++) {
            if (seen[k]) continue;
            const long new_dist = dist[j] + cost[i][k] - u[←
                i] - v[k];
            if (dist[k] > new_dist) {
                dist[k] = new_dist;
                dad[k] = j;
            }
        }
    } // update dual variables
    for (int k = 0; k < n; k++) {
        if (k == j || !seen[k]) continue;
        const int i = Rmate[k];
        v[k] += dist[k] - dist[j];
        u[i] -= dist[k] - dist[j];
    }
    u[s] += dist[j]; // augment along path
    while (dad[j] >= 0) {
        const int d = dad[j];
        Rmate[j] = Rmate[d];
        Lmate[Rmate[j]] = j;
        j = d;
    }
    Rmate[j] = s; Lmate[s] = j;
    mated++;
}
long value = 0;
for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

```

```

return value;
}
VVD cost;
cost.resize(n+m-1);
VI Lmate, Rmate;
MinCostMatching(cost, Lmate, Rmate)

```

## 4.6 Minimum Cost Maximum Flow

```

struct Edge {
    int u, v;
    long long cap, cost;
    Edge(int _u, int _v, long long _cap, long long ←
        _cost) {
        u = _u; v = _v; cap = _cap; cost = _cost;
    }
};
struct MinimumCostMaximumFlow{
    int n, s, t;
    long long flow, cost;
    vector<vector<int>> graph;
    vector<Edge> e;
    vector<long long> dist;
    vector<int> parent;
    MinimumCostMaximumFlow(int _n){
        // 0-based indexing
        n = _n;
        graph.assign(n, vector<int> ());
    }
    void add(int u, int v, long long cap, long long ←
        cost, bool directed = true){
        graph[u].push_back(e.size());
        e.push_back(Edge(u, v, cap, cost));
        graph[v].push_back(e.size());
        e.push_back(Edge(v, u, 0, -cost));
        if(!directed)
            add(v, u, cap, cost, true);
    }
    pair<long long, long long> getMinCostFlow(int ←
        _s, int _t){
        s = _s; t = _t;
        flow = 0, cost = 0;
        while(SPFA()){
            flow += sendFlow(t, 1LL<<62);
        }
        return make_pair(flow, cost);
    }
    bool SPFA(){
        parent.assign(n, -1);

```

```

dist.assign(n, 1LL<<62);          dist[s] = 0;
vector<int> queueTime(n, 0);       queueTime[s] = 1;
vector<bool> inqueue(n, 0);        inqueue[s] = true;
queue<int> q;                      q.push(s);
bool negativecycle = false;
while(!q.empty() && !negativecycle){
    int u = q.front(); q.pop(); inqueue[u] = false;
    for(int i = 0; i < graph[u].size(); i++){
        int eIdx = graph[u][i];
        int v = e[eIdx].v, w = e[eIdx].cost;
        if(dist[u] + w < dist[v] && cap > 0){
            dist[v] = dist[u] + w;
            parent[v] = eIdx;
            if(!inqueue[v]){
                q.push(v);
                queueTime[v]++;
                inqueue[v] = true;
                if(queueTime[v] == n+2){
                    negativecycle = true;
                    break;
                }
            }
        }
    }
}
return dist[t] != (1LL<<62);
}

long long sendFlow(int v, long long curFlow){
    if(parent[v] == -1) return curFlow;
    int eIdx = parent[v];
    int u = e[eIdx].u, w = e[eIdx].cost;
    long long f = sendFlow(u, min(curFlow, e[eIdx].cap));
    cost += f*w;
    e[eIdx].cap -= f;
    e[eIdx^1].cap += f;
    return f;
}

};
int source=2*n+1;
int sink=2*n+2;
MinimumCostMaximumFlow mcmf(id+10);
mcmf.add(source,i,1,k);
cout<<mcmf.getMinCostFlow(source,sink).second<<endl<<endl;

```

## 4.7 General Unweighted Maximum Matching (Edmonds' algorithm)

```

// Unweighted general matching.
// Vertices are numbered from 1 to V.
// G is an adjlist.
// G[x][0] contains the number of neighbours of x.
// The neighbours are then stored in G[x][1] .. G[x][G[x][0]].
// Mate[x] will contain the matching node for x.
// V and E are the number of edges and vertices.
// Slow Version (2x on random graphs) of Gabow's implementation
// of Edmonds' algorithm (O(V^3)).
const int MAXV = 250;
int G[MAXV][MAXV];
int VLabel[MAXV];
int Queue[MAXV];
int Mate[MAXV];
int Save[MAXV];
int Used[MAXV];
int Up, Down;
int V;

void ReMatch(int x, int y)
{
    int m = Mate[x]; Mate[x] = y;
    if (Mate[m] == x)
    {
        if (VLabel[x] <= V)
        {
            Mate[m] = VLabel[x];
            ReMatch(VLabel[x], m);
        }
        else
        {
            int a = 1 + (VLabel[x] - V - 1) / V;
            int b = 1 + (VLabel[x] - V - 1) % V;
            ReMatch(a, b); ReMatch(b, a);
        }
    }
}

void Traverse(int x)
{
    for (int i = 1; i <= V; i++) Save[i] = Mate[i];
    ReMatch(x, x);
    for (int i = 1; i <= V; i++)
    {

```

```

        if (Mate[i] != Save[i]) Used[i]++;
        Mate[i] = Save[i];
    }
}

void ReLabel(int x, int y)
{
    for (int i = 1; i <= V; i++) Used[i] = 0;
    Traverse(x); Traverse(y);
    for (int i = 1; i <= V; i++)
    {
        if (Used[i] == 1 && VLabel[i] < 0)
        {
            VLabel[i] = V + x + (y - 1) * V;
            Queue[Up++] = i;
        }
    }
}

// Call this after constructing G
void Solve()
{
    for (int i = 1; i <= V; i++)
        if (Mate[i] == 0)
        {
            for (int j = 1; j <= V; j++) VLabel[j] = ←  

                -1;
            VLabel[i] = 0; Down = 1; Up = 1; Queue[Up←  

                ++] = i;
            while (Down != Up)
            {
                int x = Queue[Down++];
                for (int p = 1; p <= G[x][0]; p++)
                {
                    int y = G[x][p];
                    if (Mate[y] == 0 && i != y)
                    {
                        Mate[y] = x; ReMatch(x, y);
                        Down = Up; break;
                    }
                }
                if (VLabel[y] >= 0)
                {
                    ReLabel(x, y);
                    continue;
                }
                if (VLabel[Mate[y]] < 0)
                {
                    VLabel[Mate[y]] = x;
                    Queue[Up++] = Mate[y];
                }
            }
        }
}

```

```

}

// Call this after Solve(). Returns number of edges←  

// in matching (half the number of matched ←  

// vertices)
int Size()
{
    int Count = 0;
    for (int i = 1; i <= V; i++)
        if (Mate[i] > i) Count++;
    return Count;
}

```

## 4.8 König's Theorem (Text)

In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. To exhibit the vertex cover:

1. Find a maximum matching
2. Change each edge **used** in the matching into a directed edge from **right to left**
3. Change each edge **not used** in the matching into a directed edge from **left to right**
4. Compute the set  $T$  of all vertices reachable from unmatched vertices on the left (including themselves)
5. The vertex cover consists of all vertices on the right that are **in**  $T$ , and all vertices on the left that are **not in**  $T$

## 4.9 Minimum Edge Cover (Text)

If a minimum edge cover contains  $C$  edges, and a maximum matching contains  $M$  edges, then  $C + M = |V|$ . To obtain the edge cover, start with a maximum matching, and then, for every vertex not matched, just select some edge incident upon it and add it to the edge set.

## 5 Data Structures

### 5.1 Persistent Segment Tree

```
struct node{
    int coun;
    node *l,*r ;
    node(int coun,node *l,node *r):
        coun(coun),l(l),r(r){}
    node *inser(int l,int r,int pos) ;
};
node* node::inser(int l,int r,int pos){
    if(l<=pos && pos<=r){
        if(l==r){
            return new node(this->coun+1,NULL,NULL);
        }
        int mid=(l+r)>>1;
        return new node(this->coun+1,this->l->inser(l,mid,
            pos),this->r->inser(mid+1,r,pos));
    }
    return this;
}
int query(node *lef,node *rig,int cc,int s,int e){
    if(s==e)
        return s;
    int co=rig->l->coun-lef->l->coun;
    int mid=(s+e)>>1;
    if(co>=cc)
        return query(lef->l,rig->l,cc,s,mid);
    return query(lef->r,rig->r,cc-co,mid+1,e);
}
node *null=new node(0,NULL,NULL);
node *root[100100];
map<int,int> m;
int mm[100100];
int arr[100100];
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    int n,mmm;cin>>n>>mmm;
    null->l=null->r=null;
    root[0]=null;
    for(int i=1;i<=n;i++) cin>>arr[i],m[arr[i]]=1 ;
    int maxy=-1;
    for(auto itr:m){
        m[itr.x]=++maxy;
        mm[maxy]=itr.x;
    }
    for(int i=1;i<=n;i++)
        root[i]=root[i-1]->inser(0,maxy,m[arr[i]]);
```

```
while(mmm-->0){
    int i,j,k;cin>>i>>j>>k;
    cout<<mm[query(root[i-1],root[j],k,0,maxy)]<<"\n"↵
    ;
}
return 0;
}
```

### 5.2 BIT- Point Update + Range Sum

```
// Binary indexed tree supporting binary search.
struct BIT {
    int n;
    vector<int> bit;
    // BIT can be thought of as having entries f[1], ..., f[n]
    // which are 0-initialized
    BIT(int n):n(n), bit(n+1) {}
    // returns f[1] + ... + f[idx-1]
    // precondition idx <= n+1
    int read(int idx) {
        idx--;
        int res = 0;
        while (idx > 0) {
            res += bit[idx];
            idx -= idx & -idx;
        }
        return res;
    }
    // returns f[idx1] + ... + f[idx2-1]
    // precondition idx1 <= idx2 <= n+1
    int read2(int idx1, int idx2) {
        return read(idx2) - read(idx1);
    }
    // adds val to f[idx]
    // precondition 1 <= idx <= n (there is no ↵
    // element 0!)
    void update(int idx, int val) {
        while (idx <= n) {
            bit[idx] += val;
            idx += idx & -idx;
        }
    }
    // returns smallest positive idx such that read↵
    // (idx) >= target
    int lower_bound(int target) {
        if (target <= 0) return 1;
        int pwr = 1; while (2*pwr <= n) pwr*=2;
```

```

    int idx = 0; int tot = 0;
    for (; pwr; pwr >>= 1) {
        if (idx+pwr > n) continue;
        if (tot + bit[idx+pwr] < target) {
            tot += bit[idx+pwr];
        }
    }
    return idx+2;
}
// returns smallest positive idx such that read<=
(idx) > target
int upper_bound(int target) {
    if (target < 0) return 1;
    int pwr = 1; while (2*pwr <= n) pwr*=2;
    int idx = 0; int tot = 0;
    for (; pwr; pwr >>= 1) {
        if (idx+pwr > n) continue;
        if (tot + bit[idx+pwr] <= target) {
            tot += bit[idx+pwr];
        }
    }
    return idx+2;
}
};

```

### 5.3 BIT- Range Update + Range Sum

```

// BIT with range updates, inspired by Petr ←
Mitrichev
struct BIT {
    int n;
    vector<int> slope;
    vector<int> intercept;
    // BIT can be thought of as having entries f←
    [1], ..., f[n]
    // which are 0-initialized
    BIT(int n): n(n), slope(n+1), intercept(n+1) {}
    // returns f[1] + ... + f[idx-1]
    // precondition idx <= n+1
    int query(int idx) {
        int m = 0, b = 0;
        for (int i = idx-1; i > 0; i -= i&-i) {
            m += slope[i];
            b += intercept[i];
        }
        return m*idx + b;
    }
    // adds amt to f[i] for i in [idx1, idx2)

```

```

    // precondition 1 <= idx1 <= idx2 <= n+1 (you ←
    can't update element 0)
    void update(int idx1, int idx2, int amt) {
        for (int i = idx1; i <= n; i += i&-i) {
            slope[i] += amt;
            intercept[i] -= idx1*amt;
        }
        for (int i = idx2; i <= n; i += i&-i) {
            slope[i] -= amt;
            intercept[i] += idx2*amt;
        }
    }
};
update(ft, p, v):
    for (; p <= N; p += p&(-p))
        ft[p] += v

# Add v to A[a...b]
update(a, b, v):
    update(B1, a, v)
    update(B1, b + 1, -v)
    update(B2, a, v * (a-1))
    update(B2, b + 1, -v * b)

query(ft, b):
    sum = 0
    for (; b > 0; b -= b&(-b))
        sum += ft[b]
    return sum

# Return sum A[1...b]
query(b):
    return query(B1, b) * b - query(B2, b)

# Return sum A[a...b]
query(a, b):
    return query(b) - query(a-1)

```

### 5.4 BIT- 2D

```

void update(int x, int y, int val){
    while (x <= max_x){
        updatey(x, y, val);
        // this function should update array tree[x←
        ]
        x += (x & -x);
    }
}

```

```

void updatey(int x , int y , int val){
    while (y <= max_y){
        tree[x][y] += val;
        y += (y & -y);
    }
}

void update(int x , int y , int val){
    int y1;
    while (x <= max_x){
        y1 = y;
        while (y1 <= max_y){
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}

int getSum(int BIT[][N+1], int x, int y)
{
    int sum = 0;
    for(; x > 0; x -= x&-x)
    {
        // This loop sum through all the 1D BIT
        // inside the array of 1D BIT = BIT[x]
        for(; y > 0; y -= y&-y)
        {
            sum += BIT[x][y];
        }
    }
    return sum;
}

```

## 6 Math

### 6.1 Convex Hull

```

struct point{
    int x, y;
    point(int _x = 0, int _y = 0){
        x = _x, y = _y;
    }
    friend bool operator < (point a, point b){
        return (a.x == b.x) ? (a.y < b.y) : (a.x < b.x);
    }
}

```

```

};
point pt[2*Max], hull[2*Max];
//Here idx is the new length of the hull
int idx=0, cur;
inline long area(point a, point b, point c){
    return (b.x - a.x) * 1LL * (c.y - a.y) - (b.y - a.y) * 1LL * (c.x - a.x);
}
inline long dist(point a, point b){
    return (a.x - b.x) * 1LL * (a.x - b.x) + (a.y - b.y) * 1LL * (a.y - b.y);
}
inline bool is_right(point a, point b){
    int dx = (b.x - a.x);
    int dy = (b.y - a.y);
    return (dx > 0) || (dx == 0 && dy > 0);
}
inline bool compare(point b, point c){
    long det = area(pt[1], b, c);
    if(det == 0){
        if(is_right(pt[1], b) != is_right(pt[1], c))
            return is_right(pt[1], b);
        return (dist(pt[1], b) < dist(pt[1], c));
    }
    return (det > 0);
}

void convexHull(){
    int min_x = pt[1].x, min_y = pt[1].y, min_idx = 1;
    for(int i = 2; i <= cur; i++){
        if(pt[i].y < min_y || (pt[i].y == min_y && pt[i].x < min_x)){
            min_x = pt[i].x;
            min_y = pt[i].y;
            min_idx = i;
        }
    }
    swap(pt[1], pt[min_idx]);
    sort(pt + 2, pt + 1 + cur, compare);
    idx = 2;
    hull[1] = pt[1], hull[2] = pt[2];
    for(int i = 3; i <= cur; i++){
        while(idx >= 2 && (area(hull[idx - 1], hull[idx], pt[i]) <= 0)) idx--;
        hull[++idx] = pt[i];
    }
}

```

### 6.2 FFT



```

// "root" is the primitive root such that
//  $\text{root}^n = 1$  modulo mod, where  $n = 2^k$  and
// root_i is the inverse of root
// FFT function takes an array L and a boolean
// parameter invert which tells whether to take
// inverse fourier transform or not, gives a new
// array which is the fourier/inverse transform of  $\leftarrow$ 
L, in
// the modulo field mod, the size of the array is
// n, which is a perfect power of 2.
// Note the transform is NOT inplace
long[] FFT(long L[], boolean invert){
    int n = L.length ;
    L = Arrays.copyOf(L, n) ;
    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){
            long tmp = L[i] ;
            L[i] = L[j] ; L[j] = tmp ;
        }
    }
    for(int m = 2 ; m <= n ; m <= 1){
        long wlen = invert ? root_i : root ;
        for(long i = m ; i < n ; i <= 1)
            wlen = (wlen * wlen) % mod ;
        long w = 1 ;
        for(int i = 0 ; i < m/2 ; i++){
            for(int k = i ; k < n ; k += m){
                long u = L[k] ;
                long v = (w * L[k + m/2]) % mod ;
                L[k] = (u + v) % mod ;
                L[k + m/2] = (u - v + mod) % mod ;
            }
            w = (w * wlen) % mod ;
        }
    }
    if(invert){
        long ninv = Mod.d(1, n) ;
        for(int i = 0 ; i < n ; i++)
            L[i] = (L[i] * ninv) % mod ;
    }
    return L ;
}

```

## 6.3 Convex Hull Trick

```

mylist hull(mylist pts){
    int n = pts.size() ;
    if(n < 2) return pts ;
    Collections.sort(pts, new Comparator<pair>(){
        public int compare(pair p1, pair p2){
            if(p1.x != p2.x) return Double.compare(p1.x, p2.x)  $\leftarrow$ 
            ;
            return Double.compare(p2.y, p1.y) ;
        }
    }) ;
    mylist h = new mylist() ;
    h.add(pts.get(0)) ; h.add(pts.get(1)) ;
    int idx = 1 ;
    for(int i = 2 ; i < n ; i++){
        pair p = pts.get(i) ;
        while(idx > 0){
            if(isOriented(h.get(idx-1), h.get(idx), p))
                break ;
            else
                h.remove(idx--) ;
        }
        h.add(p) ;
        idx++ ;
    }
    while(idx > 0 && h.get(idx).x == h.get(idx-1).x) h. $\leftarrow$ 
        remove(idx--) ;
    Collections.reverse(h) ;
    return h ;
}

public boolean isOriented(pair p1, pair p2, pair p3){
    double val = ((p2.y - p1.y) * (p3.x - p2.x)) - ((p2.x - p1.x  $\leftarrow$ 
        ) * (p3.y - p2.y)) ;
    return val >= 0 ;
}

```

## 6.4 Miscellaneous Geometry

```

// C++ routines for computational 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) {
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
// if the projection doesn't lie on the segment, ←
returns closest vertex
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)←
));
}

// 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);
}

// determine if c and d are on same side of line ←
passing through a and b
bool OnSameSide(PT a, PT b, PT c, PT d) {
return cross(c-a, c-b) * cross(d-a, d-b) > 0;
}

// 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;
}
```

## 6.5 Gaussian elimination for square matrices of full rank; finds inverses and determinants

```
// 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
//             A MUST BE INVERTIBLE!
//
// OUTPUT:     X      = an nxm matrix (stored in b↵
//             [][])
//             A^{-1} = an nxn matrix (stored in a↵
//             [][])
//             returns determinant of a[][]
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) { return 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;
}
```

## 7 Number Theory Reference

### 7.1 Fast factorization (Pollard rho) and primality testing (Rabin–Miller)

```

typedef long long unsigned int llui;
typedef long long int lli;
typedef long double float64;

llui mul_mod(llui a, llui b, llui m){
    llui y = (llui)((float64)a*(float64)b/m+(float64)1/2);
    y = y * m;
    llui x = a * b;
    llui r = x - y;
    if ( (lli)r < 0 ){
        r = r + m; y = y - 1;
    }
    return r;
}

llui C,a,b;
llui gcd(){
    llui c;
    if(a>b){
        c = a; a = b; b = c;
    }
    while(1){
        if(a == 1LL) return 1LL;
        if(a == 0 || a == b) return b;
        c = a; a = b%a;
        b = c;
    }
}

llui f(llui a, llui b){
    llui tmp;
    tmp = mul_mod(a,a,b);
    tmp+=C; tmp%=b;
    return tmp;
}

llui pollard(llui n){
    if(!(n&1)) return 2;
    C=0;
    llui iteracoes = 0;
    while(iteracoes <= 1000){
        llui x,y,d;
        x = y = 2; d = 1;
        while(d == 1){
            x = f(x,n);
            y = f(f(y,n),n);
            llui m = (x>y)?(x-y):(y-x);
            a = m; b = n; d = gcd();
        }
        if(d != n)
            return d;
        iteracoes++; C = rand();
    }
}

```

```

    return 1;
}

llui pot(llui a, llui b, llui c){
    if(b == 0) return 1;
    if(b == 1) return a%c;
    llui resp = pot(a,b>>1,c);
    resp = mul_mod(resp,resp,c);
    if(b&1)
        resp = mul_mod(resp,a,c);
    return resp;
}

// Rabin-Miller primality testing algorithm
bool isPrime(llui n){
    llui d = n-1;
    llui s = 0;
    if(n <= 3 || n == 5) return true;
    if(!(n&1)) return false;
    while(!(d&1)){ s++; d>>=1; }
    for(llui i = 0; i<32; i++){
        llui a = rand();
        a <<= 32;
        a+=rand();
        a%=(n-3); a+=2;
        llui x = pot(a,d,n);
        if(x == 1 || x == n-1) continue;
        for(llui j = 1; j<= s-1; j++){
            x = mul_mod(x,x,n);
            if(x == 1) return false;
            if(x == n-1) break;
        }
        if(x != n-1) return false;
    }
    return true;
}

map<llui,int> factors;
// Precondition: factors is an empty map, n is a ←
// positive integer
// Postcondition: factors[p] is the exponent of p ←
// in prime factorization of n
void fact(llui n){
    if(!isPrime(n)){
        llui fac = pollard(n);
        fact(n/fac); fact(fac);
    }else{
        map<llui,int>::iterator it;
        it = factors.find(n);
        if(it != factors.end()){
            (*it).second++;
        }else{
            factors[n] = 1;
        }
    }
}

```

```

    }
}

```

## 7.2 Modular arithmetic and linear Diophantine solver

```

// This is a collection of useful code for solving ←
// problems that
// involve modular linear equations. Note that all ←
// of the
// algorithms described here work on nonnegative ←
// integers.

typedef vector<int> VI;
typedef pair<int,int> PII;

// return a % b (positive value)
int mod(int a, int b) {
    return ((a%b)+b)%b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
    int tmp;
    while(b){a%=b; tmp=a; a=b; b=tmp;}
    return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
    return a/gcd(a,b)*b;
}

// returns d = gcd(a,b); finds x,y such that d = ax ←
// + by
int extended_euclid(int a, int b, int &x, int &y) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a/b;
        int t = b; b = a%b; a = t;
        t = xx; xx = x-q*xx; x = t;
        t = yy; yy = y-q*yy; y = t;
    }
    return a;
}

// finds all solutions to ax = b (mod n)

```

```

VI modular_linear_equation_solver(int a, int b, int ←
n) {
    int x, y;
    VI solutions;
    int d = extended_euclid(a, n, x, y);
    if (!(b%d)) {
        x = mod (x*(b/d), n);
        for (int i = 0; i < d; i++)
            solutions.push_back(mod(x + i*(n/d), n));
    }
    return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 ←
// on failure
int mod_inverse(int a, int n) {
    int x, y;
    int d = extended_euclid(a, n, x, y);
    if (d > 1) return -1;
    return mod(x,n);
}

// 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.first, ret. ←
second, 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;
    }
}
```

### 7.3 Polynomial Coefficients (Text)

$$(x_1 + x_2 + \dots + x_k)^n = \sum_{c_1+c_2+\dots+c_k=n} \frac{n!}{c_1!c_2!\dots c_k!} x_1^{c_1} x_2^{c_2} \dots x_k^{c_k}$$

### 7.4 Möbius Function (Text)

$$\mu(n) = \begin{cases} 0 & n \text{ not squarefree} \\ 1 & n \text{ squarefree w/ even no. of prime factors} \\ -1 & n \text{ squarefree w/ odd no. of prime factors} \end{cases}$$

Note that

$$\mu(a)\mu(b) = \mu(ab) \text{ for } a, b \text{ relatively prime}$$

Also  $\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$

**Möbius Inversion** If  $g(n) = \sum_{d|n} f(d)$  for all  $n \geq 1$ , then  $f(n) = \sum_{d|n} \mu(d)g(n/d)$  for all  $n \geq 1$ .

### 7.5 Burnside's Lemma (Text)

The number of orbits of a set  $X$  under the group action  $G$  equals the average number of elements of  $X$  fixed by the elements of  $G$ .

Here's an example. Consider a square of  $2n$  times  $2n$  cells. How many ways are there to color it into  $X$  colors, up to rotations and/or reflec-

tions? Here, the group has only 8 elements (rotations by 0, 90, 180 and 270 degrees, reflections over two diagonals, over a vertical line and over a horizontal line). Every coloring stays itself after rotating by 0 degrees, so that rotation has  $X^{4n^2}$  fixed points. Rotation by 180 degrees and reflections over a horizontal/vertical line split all cells in pairs that must be of the same color for a coloring to be unaffected by such rotation/reflection, thus there exist  $X^{2n^2}$  such colorings for each of them. Rotations by 90 and 270 degrees split cells in groups of four, thus yielding  $X^{n^2}$  fixed colorings. Reflections over diagonals split cells into  $2n$  groups of 1 (the diagonal itself) and  $2n^2 - n$  groups of 2 (all remaining cells), thus yielding  $X^{2n^2-n+2n} = X^{2n^2+n}$  unaffected colorings. So, the answer is  $(X^{4n^2} + 3X^{2n^2} + 2X^{n^2} + 2X^{2n^2+n})/8$ .

## 8 Miscellaneous

### 8.1 2-SAT

```
// 2-SAT solver based on Kosaraju's algorithm.
// Variables are 0-based. Positive variables are ←
// stored in vertices 2n, corresponding negative ←
// variables in 2n+1
// TODO: This is quite slow (3x-4x slower than ←
// Gabow's algorithm)
struct TwoSat {
    int n;
    vector<vector<int>> adj, radj, scc;
    vector<int> sid, vis, val;
    stack<int> stk;
    int scnt;

    // n: number of variables, including negations
    TwoSat(int n): n(n), adj(n), radj(n), sid(n), vis(←
        n), val(n, -1) {}

    // adds an implication
    void impl(int x, int y) { adj[x].push_back(y); ←
        radj[y].push_back(x); }

    // adds a disjunction
    void vee(int x, int y) { impl(x^1, y); impl(y^1, x ←
        ); }

    // forces variables to be equal
```



```

void eq(int x, int y) { impl(x, y); impl(y, x); ←
    impl(x^1, y^1); impl(y^1, x^1); }
// forces variable to be true
void tru(int x) { impl(x^1, x); }

void dfs1(int x) {
    if (vis[x]++) return;
    for (int i = 0; i < adj[x].size(); i++) {
        dfs1(adj[x][i]);
    }
    stk.push(x);
}

void dfs2(int x) {
    if (!vis[x]) return; vis[x] = 0;
    sid[x] = scnt; scc.back().push_back(x);
    for (int i = 0; i < radj[x].size(); i++) {
        dfs2(radj[x][i]);
    }
}

// returns true if satisfiable, false otherwise
// on completion, val[x] is the assigned value of ←
// variable x
// note, val[x] = 0 implies val[x^1] = 1
bool two_sat() {
    scnt = 0;
    for (int i = 0; i < n; i++) {
        dfs1(i);
    }
    while (!stk.empty()) {
        int v = stk.top(); stk.pop();
        if (vis[v]) {
            scc.push_back(vector<int>());
            dfs2(v);
            scnt++;
        }
    }
    for (int i = 0; i < n; i += 2) {
        if (sid[i] == sid[i+1]) return false;
    }
    vector<int> must(scnt);
    for (int i = 0; i < scnt; i++) {
        for (int j = 0; j < scc[i].size(); j++) {
            val[scc[i][j]] = must[i];
            must[sid[scc[i][j]^1]] = !must[i];
        }
    }
    return true;
}
};

```

## 8.2 Stable Marriage Problem (Gale–Shapley algorithm)

```

// Gale-Shapley algorithm for the stable marriage ←
// problem.
// madj[i][j] is the jth highest ranked woman for ←
// man i.
// fpref[i][j] is the rank woman i assigns to man j ←
// .
// Returns a pair of vectors (mpart, fpart), where ←
// mpart[i] gives the partner of man i, and fpart ←
// is analogous
pair<vector<int>, vector<int>> stable_marriage(←
    vector<vector<int>> &madj, vector<vector<int>> ←
    &fpref) {
    int n = madj.size();
    vector<int> mpart(n, -1), fpart(n, -1);
    vector<int> midx(n);
    queue<int> mfree;
    for (int i = 0; i < n; i++) {
        mfree.push(i);
    }
    while (!mfree.empty()) {
        int m = mfree.front(); mfree.pop();
        int f = madj[m][midx[m]++];
        if (fpart[f] == -1) {
            mpart[m] = f; fpart[f] = m;
        } else if (fpref[f][m] < fpref[f][fpart[f]]) {
            mpart[fpart[f]] = -1; mfree.push(fpart[f]);
            mpart[m] = f; fpart[f] = m;
        } else {
            mfree.push(m);
        }
    }
    return make_pair(mpart, fpart);
}

```

## 9 Credits

1. BrianBi for Codebook Latex and some code snippets.
2. Animesh Fatehpuria for Code snippets.