**Walsh Hadamard Transformation:**

**struct** Walsh\_Hadamard{  
  
 **long** **long** P1[MAX], P2[MAX];  
  
 **void** walsh\_transform(**long** **long**\* ar, **int** n, **int** flag , **int** invFlag){  
 **if** (n == 0) **return**;  
  
 **int** i, m = n / 2;  
 walsh\_transform(ar, m, flag, invFlag);  
 walsh\_transform(ar + m, m, flag, invFlag);  
  
 **for** (i = 0; i < m; i++){ */// Don't forget modulo if required*  
 **long** **long** x = ar[i], y = ar[i + m];  
 **if** (flag == OR) ar[i] = x, ar[i + m] = invFlag\*x + y;  
 **if** (flag == AND) ar[i] = x + invFlag\*y, ar[i + m] = y;  
 **if** (flag == XOR){  
 ar[i] = x + y, ar[i + m] = x - y;  
 **if**(invFlag==-1) ar[i]>>=1, ar[i+m]>>=1;  
 }  
 }  
 }  
  
 vector <**long** **long**> convolution(**int** n, **long** **long**\* A, **long** **long**\* B, **int** flag = XOR){  
 assert(\_\_builtin\_popcount(n) == 1); */// n must be a power of 2*  
 **for** (**int** i = 0; i < n; i++) P1[i] = A[i];  
 **for** (**int** i = 0; i < n; i++) P2[i] = B[i];  
  
 walsh\_transform(P1, n, flag, 1);  
 walsh\_transform(P2, n, flag, 1);  
 **for** (**int** i = 0; i < n; i++) P1[i] = P1[i] \* P2[i];  
 walsh\_transform(P1, n, flag, -1);  
 **return** vector<**long** **long**> (P1, P1 + n);  
 }  
  
 */// For i = 0 to n - 1, j = 0 to n - 1*  
 */// v[i or j] += A[i] \* B[j]*  
 vector <**long** **long**> or\_convolution(**int** n, **long** **long**\* A, **long** **long**\* B){  
 **return** convolution(n, A, B, OR);  
 }  
  
 */// For i = 0 to n - 1, j = 0 to n - 1*  
 */// v[i and j] += A[i] \* B[j]*  
 vector <**long** **long**> and\_convolution(**int** n, **long** **long**\* A, **long** **long**\* B){  
 **return** convolution(n, A, B, AND);  
 }  
  
 */// For i = 0 to n - 1, j = 0 to n - 1*  
 */// v[i xor j] += A[i] \* B[j]*  
 vector <**long** **long**> xor\_convolution(**int** n, **long** **long**\* A, **long** **long**\* B){  
 **return** convolution(n, A, B, XOR);  
 }  
  
}wh;

**Sudoku Solver:**

*/\**  
 *Input: Char\*\* str (A 2D character array containing the initial*  
 *configuration of the board)*  
  
 *Define EMPTY\_CELL as the character that defines an empty cell in str*  
  
 *Call sudoku(char\*\* str) to initialize the sudoku structure*  
  
 *Call solve() to find a solution to the puzzle (May return any solution*  
 *if not unique)*  
  
 *solve() returns true if a solution exits, false otherwise.*  
 *If true is returned, var[][] will contain the values of each cell*  
  
 *Call print() to print the solved puzzle (Adjust according to problem statement)*  
*\*/*  
  
#define EMPTY\_CELL '.'  
**typedef** pair <**int**, **int**> pii;  
#define XX first  
#define YY second  
**struct** sudoku{  
 **int** var[15][15];  
 **int** dom[15][15], pdom[15][15];  
 pii pos[15][15];  
 **int** pcnt[15];  
  
 **int** group[10][10]={  
 {0, 0, 0, 1, 1, 1, 2, 2, 2},  
 {0, 0, 0, 1, 1, 1, 2, 2, 2},  
 {0, 0, 0, 1, 1, 1, 2, 2, 2},  
 {3, 3, 3, 4, 4, 4, 5, 5, 5},  
 {3, 3, 3, 4, 4, 4, 5, 5, 5},  
 {3, 3, 3, 4, 4, 4, 5, 5, 5},  
 {6, 6, 6, 7, 7, 7, 8, 8, 8},  
 {6, 6, 6, 7, 7, 7, 8, 8, 8},  
 {6, 6, 6, 7, 7, 7, 8, 8, 8}  
 }; *//Sub-sections that cannot have the same number are marked with same grouo[][] value*  
  
 **inline** **int** set\_bit(**int** mask, **int** pos){ **return** (mask | (1<<pos)); }  
 **inline** **int** reset\_bit(**int** mask, **int** pos){ **return** (mask & ~(1<<pos)); }  
 **inline** **bool** check\_bit(**int** mask, **int** pos){ **return** (mask & (1<<pos))!=0; }  
  
  
 **inline** **void** save(**int** pdom[15][15]){  
 **for**(**int** i=0; i<9; i++){  
 **for**(**int** j=0; j<9; j++){  
 pdom[i][j]=dom[i][j];  
 }  
 }  
 }  
  
 **inline** **void** color\_var(**int** px, **int** py, **int** color){  
 var[px][py]=color;  
 pii tmp;  
 **for**(**int** i=0; i<9; i++) dom[px][i]=reset\_bit(dom[px][i], color);  
 **for**(**int** i=0; i<9; i++) dom[i][py]=reset\_bit(dom[i][py], color);  
 **for**(**int** i=0; i<9; i++){  
 tmp=pos[group[px][py]][i];  
 dom[tmp.XX][tmp.YY]=reset\_bit(dom[tmp.XX][tmp.YY], color);  
 }  
 dom[px][py]=0;  
 dom[px][py]=set\_bit(dom[px][py], color);  
 }  
  
 **inline** **void** restore(**int** pdom[15][15]){  
 **for**(**int** i=0; i<9; i++){  
 **for**(**int** j=0; j<9; j++){  
 dom[i][j]=pdom[i][j];  
 }  
 }  
 }  
  
 **inline** pii findNextPair(){ */\*(-1, -1) if invalid,*  
 *(-2,-2) if solved,*  
 *(x, y) if (x, y) is most constrained \*/*  
 **bool** sflag=**true**, fg=**false**;  
 pii mn;  
 **int** cnt;  
 **for**(**int** i=0; i<9; i++){  
 **for**(**int** j=0; j<9; j++){  
 cnt=\_\_builtin\_popcount(dom[i][j]);  
 **if**(cnt==0){  
 **return** pii(-1,-1);  
 }  
 **if**(cnt!=1) {  
 sflag=**false**;  
 **if**(!fg || cnt<(\_\_builtin\_popcount(dom[mn.XX][mn.YY]))){  
 fg=**true**;  
 mn=pii(i, j);  
 }  
 }  
 **else**{  
 **for**(**int** k=1; k<=9; k++){  
 **if**(check\_bit(dom[i][j], k)){  
 color\_var(i, j, k);  
 **break**;  
 }  
 }  
 }  
 }  
 }  
 **if**(sflag) **return** pii(-2, -2);  
 **return** mn;  
 }  
  
 sudoku(){}  
 sudoku(**char** str[15][15]){  
 memset(pcnt, 0, **sizeof**(pcnt));  
 **int** val=0;  
 **for**(**int** i=1; i<=9; i++){  
 val=set\_bit(val, i);  
 }  
 **for**(**int** i=0; i<9; i++){  
 **for**(**int** j=0; j<9; j++){  
 pos[group[i][j]][pcnt[group[i][j]]]=pii(i, j);  
 pcnt[group[i][j]]++;  
 dom[i][j]=val;  
 }  
 }  
 **for**(**int** i=0; i<9; i++){  
 **for**(**int** j=0; j<9; j++){  
 **if**(str[i][j]!=EMPTY\_CELL){  
 color\_var(i, j, str[i][j]-'0');  
 }  
 }  
 }  
 }  
  
 **bool** backtrack(**int** cx, **int** cy){  
 **int** pdom[15][15];  
 save(pdom);  
 pii tmp;  
 **for**(**int** i=1; i<=9; i++){  
 **if**(check\_bit(dom[cx][cy], i)){  
 color\_var(cx, cy, i);  
 tmp=findNextPair();  
 **if**(tmp==pii(-2, -2)) **return** **true**;  
 **else** **if**(tmp==pii(-1, -1)) restore(pdom);  
 **else**{  
 **if**(backtrack(tmp.XX, tmp.YY)) **return** **true**;  
 restore(pdom);  
 }  
 }  
 }  
 **return** **false**;  
 }

**Primes For Hashing:**

500000003 500002597

1000002307 1000019903

**Pick’s Theorem:**

/\*

Input: point[] para, polygon with vertices sorted in CW or CCW order

Output: Number of lattice points strictly inside polygon

\*/

#define SIZE 1005

typedef long long ll;

struct point{

ll x, y;

}para[SIZE];

int n;

ll areaOfPolygon2(){ //2\*Area of simple polygon para[], which has n points

para[n]=para[0];

ll area=0;

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

area+=(para[i].x)\*(para[i+1].y);

area-=(para[i+1].x)\*(para[i].y);

}

if(area<0LL) area\*=-1; //If points are in clockwise order

return area;

}

ll numberOfLatticePoints(point a, point b){ //Number of lattice points in line segment connecting a, b

ll xd=a.x>b.x ? a.x-b.x : b.x-a.x;

ll yd=a.y>b.y ? a.y-b.y : b.y-a.y;

return \_\_gcd(xd, yd); //All lattice points except one vertex

}

ll latticePointsInsidePoly(){ //Number of lattice points strictly inside polygon

ll area=areaOfPolygon2();

ll b=0;

for(int i=0; i<n; i++) b+=numberOfLatticePoints(para[i], para[i+1]);

return (area-b)/2+1;

}

**Min Cost Max Flow:**

**int** src, snk, nNode, nEdge;  
**int** fin[MAXN], pre[MAXN], dist[MAXN];  
**int** cap[MAXE], cost[MAXE], nxt[MAXE], to[MAXE], from[MAXE];

**inline** **void** init(**int** \_src, **int** \_snk, **int** nodes){  
 memset(fin, -1, **sizeof**(fin));  
 nNode=nodes, nEdge=0;  
 src=\_src, snk=\_snk;  
}  
  
**inline** **void** addEdge(**int** u, **int** v, **int** \_cap, **int** \_cost){  
 from[nEdge]=u, to[nEdge]=v, cap[nEdge]=\_cap, cost[nEdge]=\_cost;  
 nxt[nEdge]=fin[u], fin[u]=nEdge++;  
 from[nEdge]=v, to[nEdge]=u, cap[nEdge]=0, cost[nEdge]=-(\_cost);  
 nxt[nEdge]=fin[v], fin[v]=nEdge++;  
}  
  
**bool** bellman(){  
 **int** iter, u, v, i;  
 **bool** flag=**true**;  
 memset(dist, 0x7f, **sizeof**(dist));  
 memset(pre, -1, **sizeof**(pre));  
 dist[src]=0;  
 **for**(iter=1; iter<nNode && flag; iter++){  
 flag=**false**;  
 **for**(u=0; u<nNode; u++){  
 **for**(i=fin[u]; i>=0; i=nxt[i]){  
 v=to[i];  
 **if**(cap[i] && dist[v]>dist[u]+cost[i]){  
 dist[v]=dist[u]+cost[i];  
 pre[v]=i;  
 flag=**true**;  
 }  
 }  
 }  
 }  
 **return** (dist[snk]<INF);  
}  
  
**int** mcmf(**int** &fcost){  
 **int** netflow, i, bot, u;  
 netflow=fcost=0;  
 **while**(bellman()){  
 bot=INF;  
 **for**(u=pre[snk]; u>=0; u=pre[from[u]]) bot=min(bot, cap[u]);  
 **for**(u=pre[snk]; u>=0; u=pre[from[u]]){  
 cap[u]-=bot;  
 cap[u^1]+=bot;  
 fcost+=bot\*cost[u];  
 }  
 netflow+=bot;  
 }  
 **return** netflow;  
}

**Maximum Point Cover in Circle of Given Radius:**

*/\**  
 *Simply call maxPointCover()*  
 *Input: p[] contains the co-ordinates*  
 *of n points.*  
 *Output: Finds the maximum number of points*  
 *covered by a circle of radius r.*  
 *Complexity: O(n^2 log n)*  
*\*/*  
#define SIZE 2005  
#define EPS 1e-9  
#define XX first  
#define YY second  
**typedef** **long** **double** ld;  
**typedef** pair < ld, **bool** > plb;  
**struct** point{  
 **int** x, y;  
 point(){}  
 point(**int** xx, **int** yy){  
 x=xx;  
 y=yy;  
 }  
}p[SIZE];  
**int** n, r;  
complex < ld > cpy[SIZE];  
ld dis[SIZE][SIZE];  
  
**void** makePointsComplex(){  
 **for**(**int** i=0; i<n; i++){  
 cpy[i]=complex < ld > (p[i].x, p[i].y);  
 }  
}  
  
**bool** cmp(plb a, plb b){  
 **if**(abs(a.XX-b.XX)<EPS) **return** a.YY>b.YY;  
 **return** a.XX<b.XX;  
}  
  
plb angles[2\*SIZE];  
  
**int** maxPointCoverWithI(**int** i, ld r, **int** n){  
 **int** anglesz=0;  
 ld b, a, alpha, beta;  
 **for**(**int** j=0; j<n; j++){  
 **if**(i!=j && dis[i][j]<=(2.0\*r+EPS)){  
 b=acos(dis[i][j]/(2.0\*r));  
 a=arg(cpy[j]-cpy[i]);  
 alpha=a-b;  
 beta=a+b;  
 angles[anglesz++]=plb(alpha, **true**);  
 angles[anglesz++]=plb(beta, **false**);  
 }  
 }  
 sort(angles, angles+anglesz, cmp);  
 **int** cnt=1, res=1;  
 **for**(**int** i=0; i<anglesz; i++){  
 **if**(angles[i].YY) cnt++;  
 **else** cnt--;  
 res=max(cnt, res);  
 }  
 **return** res;  
}  
  
**int** maxPointCover(){  
 makePointsComplex();  
 **for**(**int** i=0; i<(n-1); i++){  
 **for**(**int** j=i+1; j<n; j++){  
 dis[i][j]=dis[j][i]=abs(cpy[i]-cpy[j]);  
 }  
 }  
 **int** ans=0;  
 **for**(**int** i=0; i<n; i++){  
 ans=max(ans, maxPointCoverWithI(i, r, n));  
 }  
 **return** ans;  
}

**Infix Expression to Postfix:**

#define NUMBER\_OF\_OPERATORS 3  
**using** **namespace** std;  
  
  
**char** op[]={'\*', '.', '|'}; */\*List all operators in*  
 *decreasing order of*  
 *precedence\*/*  
  
**inline** **int** operator\_precedence(**char** ch){  
 **for**(**int** i=0; i<NUMBER\_OF\_OPERATORS; i++) **if**(ch==op[i]) **return** i;  
}  
  
**inline** **bool** isOperator(**char** ch){  
 **for**(**int** i=0; i<NUMBER\_OF\_OPERATORS; i++){  
 **if**(ch==op[i]) **return** **true**;  
 }  
 **return** **false**;  
}  
  
stack <**char**> opstack;  
  
**void** infixToPostfix(**char** postFix[], **int** &postlen, **char** infix[], **int** &inlen){  
 **while**(!opstack.empty()) opstack.pop();  
 postlen=0;  
 **for**(**int** i=0; i<inlen; i++){  
 **if**(infix[i]=='('){  
 opstack.push('(');  
 }  
 **else** **if**(infix[i]==')'){  
 **while**(opstack.top()!='('){  
 postFix[postlen]=opstack.top();  
 postlen++;  
 opstack.pop();  
 }  
 opstack.pop();  
 }  
 **else** **if**(!isOperator(infix[i])){  
 postFix[postlen]=infix[i];  
 postlen++;  
 }  
 **else**{  
 **while**(!opstack.empty() && operator\_precedence(opstack.top())<=operator\_precedence(infix[i])){  
 postFix[postlen]=opstack.top();  
 postlen++;  
 opstack.pop();  
 }  
 opstack.push(infix[i]);  
 }  
 }  
 **while**(!opstack.empty()){  
 postFix[postlen]=opstack.top();  
 postlen++;  
 opstack.pop();  
 }  
 postFix[postlen]=0;  
}

**Fast Fourier Transformation:**

*/\**  
 *Call Multiply to multiply two polynomials.*  
 *Vectors a and b hold the co-efficients of the polynomials.*  
 *Vector res will contain the co-efficients of a(x)\*b(x)*  
  
 *Call multiply\_two\_numbers, to multiply two strings*  
*\*/*  
  
#define PI acos(-1.0)  
**using** **namespace** std;  
**typedef** complex <**double**> base;  
**void** fft(vector <base> &a, **bool** invert){  
 **int** n=(**int**) a.size();  
 **if**(n==1) **return**;  
  
 vector <base> a0(n>>1), a1(n>>1);  
 **for**(**int** i=0, j=0; i<n; i+=2, j++){  
 a0[j]=a[i];  
 a1[j]=a[i+1];  
 }  
 fft(a0, invert);  
 fft(a1, invert);  
  
 **double** ang=2\*PI/n \* (invert? -1: 1);  
 base w(1), wn(cos(ang), sin(ang));  
 **for**(**int** i=0; i<(n>>1); i++){  
 a[i]=a0[i]+w\*a1[i];  
 a[i+(n>>1)]=a0[i]-w\*a1[i];  
 **if**(invert) a[i]/=2, a[i+(n>>1)]/=2;  
 w\*=wn;  
 }  
}  
  
**void** multiply(**const** vector <**int**> &a, **const** vector <**int**> &b, vector <**int**> &res){  
 vector <base> fa(a.begin(), a.end()), fb(b.begin(), b.end());  
 **int** 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**(**int** i=0; i<n; i++) fa[i]\*=fb[i];  
 fft(fa, **true**);  
 res.resize(n);  
 **for**(**int** i=0; i<n; i++) res[i]=**int**(fa[i].real()+0.5);  
}  
  
**void** multiply\_two\_numbers(**char**\* an, **char**\* bn, **int** al, **int** bl, vector <**int**> &res){  
 vector <**int**> a, b;  
 **for**(**int** i=0; i<al; i++) a.push\_back(**int**(an[i]-'0'));  
 **for**(**int** i=0; i<bl; i++) b.push\_back(**int**(bn[i]-'0'));  
 reverse(a.begin(), a.end());  
 reverse(b.begin(), b.end());  
 multiply(a, b, res);  
 **int** carry=0;  
 **for**(**int** i=0; i<(**int**)res.size(); i++){  
 res[i]+=carry;  
 carry=res[i]/10;  
 res[i]%=10;  
 }  
 **if**(carry) res.push\_back(carry);  
}

**Max Flow (Dinitz):**

/\*

Add an edge between nodes u and v of capacity w

by making res[u][v]=w and adj[u][v]=adj[v][u]=true

\*/

**int** res[NODES][NODES], nxt[NODES][NODES];  
**bool** adj[NODES][NODES];  
**int** q[NODES], lvl[NODES];  
**int** src, sink;  
**inline** **void** init(){  
 memset(res, 0, **sizeof**(res));  
 memset(nxt, -1, **sizeof**(nxt));  
 memset(adj, **false**, **sizeof**(adj));  
}  
  
**inline** **void** process\_graph(){  
 **for**(**int** i=0; i<NODES; i++){  
 **for**(**int** j=NODES-2; j>=0; j--){  
 nxt[i][j]=(adj[i][j+1]) ? j+1 : nxt[i][j+1];  
 }  
 }  
}  
  
**bool** bfs(){  
 memset(lvl, -1, **sizeof**(lvl));  
 lvl[src]=0;  
 **int** st=0, ed=1;  
 q[st]=src;  
 **int** u, v;  
 **while**(st!=ed){  
 u=q[st];  
 st++;  
 v=(adj[u][0]) ? 0:nxt[u][0];  
 **for**(; v!=-1; v=nxt[u][v]){  
 **if**(lvl[v]==-1 && res[u][v]){  
 q[ed]=v;  
 lvl[v]=lvl[u]+1;  
 ed++;  
 }  
 }  
 }  
 **return** (lvl[sink]!=-1);  
}  
  
**int** dfs(**int** u, **int** flow){  
 **if**(u==sink) **return** flow;  
 **int** v=(adj[u][0])? 0:nxt[u][0];  
 **int** tmp;  
 **for**(; v!=-1; v=nxt[u][v]){  
 **if**(res[u][v] && lvl[v]==lvl[u]+1){  
 tmp=dfs(v, min(flow, res[u][v]));  
 **if**(tmp){  
 res[u][v]-=tmp;  
 res[v][u]+=tmp;  
 **return** tmp;  
 }  
 }  
 }  
 **return** 0;  
}  
  
**int** max\_flow(){  
 process\_graph();  
 **int** res=0, tmp;  
 **while**(bfs()){  
 **while**((tmp=dfs(src, INF))){  
 res+=tmp;  
 }  
 }  
 **return** res;  
}

**Diameter of a Tree (And Related):**

#define NODES 100105  
*/\**  
 *\* Assumes 0 based indexing of tree*  
 *\* Use clearAll for multiple test cases*  
 *\* Call proc() to calculate the tree's diametre,*  
 *\* nodes at the ends of the diameter,*   
 *\* and nodes at the centre of the diameter*   
*\*/*  
**struct** Tree{  
 vector <**int**> adj[NODES], centre; *//Adjacency List and nodes at centre of diameter respectively*  
 vector <**int**> adjw[NODES]; *//Adjacence List of edge weights*  
 **int** dis[3][NODES];  
 **int** far[NODES]; *//Distance of farthest node from each node*  
 **int** end1, end2, n, dia; *//n=number of vertices of tree*  
 *//end1 and end2 are the nodes at the diameter;*  
 *//dia is the diametre of the tree*  
 **void** clearAll(){  
 centre.clear();  
 **for**(**int** i=0; i<NODES; i++){  
 adj[i].clear();  
 adjw[i].clear();  
 }  
 }  
 **void** addEdge(**int** u, **int** v, **int** w){ *//Adds edge between nodes u, v with weight w*  
 adj[u].push\_back(v);  
 adjw[u].push\_back(w);  
 adj[v].push\_back(u);  
 adjw[v].push\_back(w);  
 }  
 **void** dfs(**int** nd, **int** ind){  
 **int** v;  
 **for**(**int** i=0; i<adj[nd].size(); i++){  
 v=adj[nd][i];  
 **if**(dis[ind][v]==-1){  
 dis[ind][v]=adjw[nd][i]+dis[ind][nd];  
 dfs(v, ind);  
 }  
 }  
 }  
 **int** findmax(**int** ind){  
 **int** mx=0;  
 **for**(**int** i=1; i<n; i++){  
 **if**(dis[ind][mx]<dis[ind][i]) mx=i;  
 }  
 **return** mx;  
 }  
 **void** findDia(){  
 memset(dis, -1, **sizeof**(dis));  
 dis[0][0]=0;  
 dfs(0, 0);  
 end1=findmax(0);  
 dis[1][end1]=0;  
 dfs(end1, 1);  
 end2=findmax(1);  
 dis[2][end2]=0;  
 dfs(end2, 2);  
 dia=dis[1][end2];  
 }  
 **void** findFarthestNodes(){  
 **for**(**int** i=0; i<n; i++){  
 far[i]=max(dis[1][i], dis[2][i]);  
 }  
 }  
 **int** findMinFar(){  
 **int** mn=far[0];  
 **for**(**int** i=1; i<n; i++){  
 mn=min(mn, far[i]);  
 }  
 **return** mn;  
 }  
 **void** findCentres(){  
 **int** mn=findMinFar();  
 **for**(**int** i=0; i<n; i++){  
 **if**(far[i]==mn){  
 centre.push\_back(i);  
 }  
 }  
 }  
 **void** proc(){  
 findDia();  
 findFarthestNodes();  
 findCentres();  
 }  
};