**Data Structures**

***Union–Find Disjoint Sets***

// NOTE: If there are v vertexes, their

// indexes are numbered from 0 to v - 1.

struct DisjointSet**{**

int sets**;**

int**\*** parent**;**

int**\*** members**;**

void init**(**int n**){** // O(v)

sets **=** n**;**

parent **=** **new** int**[**n**];**

members **=** **new** int**[**n**];**

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

parent**[**i**]** **=** i**,** members**[**i**]** **=** 1**;**

**}**

int find**(**int a**){** // O(1)

**if(**a **==** parent**[**a**])** **return** a**;**

**return** parent**[**a**]** **=** find**(**parent**[**a**]);**

**}**

int setsize**(**int a**){** // O(1)

**return** members**[**find**(**a**)];**

**}**

bool joined**(**int a**,** int b**){** // O(1)

**return** find**(**a**)** **==** find**(**b**);**

**}**

void join**(**int a**,** int b**){** // O(1)

**if(**joined**(**a**,** b**))** **return;**

members**[**find**(**b**)]** **+=** members**[**find**(**a**)];**

parent**[**find**(**a**)]** **=** find**(**b**);**

sets**--;**

**}**

**};**

***Segment Tree***

// NOTE: Tree indexed from 0 to n - 1.

// If you want lazy propagation on tree

// uncomment code tagged as 'Propagation'.

// IMPORTANT: Functions are very generic,

// the way of handling values may change.

#define M ((l + r) / 2)

#define Left (node \* 2)

#define Right (node \* 2 + 1)

struct SegmentTree**{**

int**\*** tree**;**

int**\*** lazy**;**

/\* Propagation

void propagate(int node, int l, int r){ // O(1)

if(l == r) return;

lazy[Left] = lazy[node];

lazy[Right] = lazy[node];

} \*/

int create**(**int node**,** int l**,** int r**){** // O(n log n)

lazy**[**node**]** **=** 0**;**

**if(**l **==** r**)** **return** tree**[**node**]** **=** 0**;**

int L **=** create**(**Left**,** l**,** M**);**

int R **=** create**(**Right**,** M **+** 1**,** r**);**

**return** tree**[**node**]** **=** L **+** R**;**

**}**

int update**(**int node**,** int l**,** int r**,** int a**,** int set**){**

/\* Propagation

if(lazy[node] != 0){

propagate(node, l, r);

tree[node] = lazy[node];

lazy[node] = 0;

} \*/

**if(**a **<** l **||** r **<** a**)**

**return** tree**[**node**];**

**if(**a **<=** l **&&** r **<=** a**){**

/\* Propagation

lazy[node] = set;

propagate(node, l, r);

lazy[node] = 0; \*/

**return** tree**[**node**]** **=** set**;**

**}**

int L **=** update**(**Left**,** l**,** M**,** a**,** set**);**

int R **=** update**(**Right**,** M **+** 1**,** r**,** a**,** set**);**

**return** tree**[**node**]** **=** L **+** R**;**

**}** // O(log n)

int query**(**int node**,** int l**,** int r**,** int a**,** int b**){**

**if(**b **<** l **||** r **<** a**)**

**return** 0**;**

/\* Propagation

if(lazy[node] != 0){

propagate(node, l, r);

tree[node] = lazy[node];

lazy[node] = 0;

}\*/

**if(**a **<=** l **&&** r **<=** b**)**

**return** tree**[**node**];**

int L **=** query**(**Left**,** l**,** M**,** a**,** b**);**

int R **=** query**(**Right**,** M **+** 1**,** r**,** a**,** b**);**

**return** L **+** R**;**

**}** // O(log n)

void init**(**int n**){** // O(n log n)

tree **=** **new** int**[**n **\*** 4**];**

lazy **=** **new** int**[**n **\*** 4**];**

create**(**1**,** 0**,** n **-** 1**);**

**}**

**};**

***Sparse Table***

// NOTE: Array indexed from 0 to n - 1.

#define LOGN 20

#define MAXN 100000

int num**[**MAXN**];**

int logs**[**MAXN**];**

int sparse**[**MAXN**][**LOGN**];**

void precalc\_sparse**(**int n**){** // O(n log n)

**for(**int i **=** 2**;** i **<** MAXN**;** i**++)**

logs**[**i**]** **=** logs**[**i **/** 2**]** **+** 1**;**

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

sparse**[**i**][**0**]** **=** num**[**i**];**

**for(**int i **=** 0**;** i **<** logs**[**n**];** i**++){**

**for(**int j **=** 0**;** j **<** n**;** j**++){**

int k **=** min**(**j **+** **(**1 **<<** i**),** n **-** 1**);**

int join **=** max**(**sparse**[**j**][**i**],** sparse**[**k**][**i**]);**

sparse**[**j**][**i **+** 1**]** **=** join**;**

**}**

**}**

**}**

int query**(**int a**,** int b**){** // O(1)

**if(**a **==** b**)** **return** sparse**[**a**][**0**];**

int binlog **=** logs**[**b **-** a **+** 1**];**

int k **=** b **-** **(**1 **<<** binlog**)** **+** 1**;**

**return** max**(**sparse**[**a**][**binlog**],** sparse**[**k**][**binlog**]);**

**}**

**Graph Algorithms**

***Topological Sort***

// NOTE: Vertexes indexed from 0 to v - 1.

// SCC algorithm below implicitly provides

// topological sort of the contracted DAG.

#include <vector>

#include <algorithm>

**using** **namespace** std**;**

#define MAXV 100000

int sortedVertexes**[**MAXV**];** // Result

int vertexNumber**;**

bool visited**[**MAXV**];**

vector**<**int**>** E**[**MAXV**];**

void TopologicalTraverse**(**int u**){** // O(v + e)

visited**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++)**

**if(!**visited**[\***v**])** TopologicalTraverse**(\***v**);**

sortedVertexes**[**vertexNumber**++]** **=** u**;**

**}**

void TopologicalSort**(**int v**){** // O(v + e)

vertexNumber **=** 0**;**

fill**(**visited**,** visited **+** v**,** **false);**

**for(**int i **=** 0**;** i **<** v**;** i**++)**

**if(!**visited**[**i**])** TopologicalTraverse**(**i**);**

reverse**(**sortedVertexes**,** sortedVertexes **+** v**);**

**}**

***Articulation Points and Bridges***

// NOTE: Vertexes indexed from 0 to v - 1.

// Be careful using this function more than once.

// IMPORTANT: When calling the function, use the

// number of any vertex of the component and -1.

#include <set>

#include <vector>

**using** **namespace** std**;**

#define MAXV 100000

set**<**int**>** ArticulationPoints**;** // Result

set**<** pair**<**int**,** int**>** **>** Bridges**;** // Result

int low**[**MAXV**];**

int num**[**MAXV**];**

int rootChild **=** 0**;**

int vertexCount **=** 1**;**

vector**<**int**>** E**[**MAXV**];**

void ArticulationPointsAndBridges**(**int u**,** int p**){**

vector**<**int**>::**iterator v**;**

low**[**u**]** **=** num**[**u**]** **=** vertexCount**++;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++){**

**if(**num**[\***v**]** **==** 0**){**

**if(**p **==** **-**1**)** rootChild**++;**

ArticulationPointsAndBridges**(\***v**,** u**);**

**if(**num**[**u**]** **<=** low**[\***v**])**

ArticulationPoints**.**insert**(**u**);**

**if(**num**[**u**]** **<** low**[\***v**])**

Bridges**.**insert**(**make\_pair**(**u**,** **\***v**));**

low**[**u**]** **=** min**(**low**[\***v**],** low**[**u**]);**

**}** **else** **if(\***v **!=** p**)**

low**[**u**]** **=** min**(**num**[\***v**],** low**[**u**]);**

**}**

**if(**p **==** **-**1 **&&** rootChild **<** 2**)**

ArticulationPoints**.**erase**(**u**);**

**}** // O(v + e)

***Strongly Connected Components***

// NOTE: Vertexes indexed from 0 to v - 1.

// Be careful using this function more than once.

#include <stack>

#include <vector>

**using** **namespace** std**;**

#define MAXV 100000

vector**<** vector**<**int**>** **>** SCC**;** // Result

int low**[**MAXV**];**

int num**[**MAXV**];**

bool inStack**[**MAXV**];**

stack**<**int**>** Stack**;** // WARNING: This may be slow.

vector**<**int**>** E**[**MAXV**];**

int vertexCount **=** 1**;**

void StronglyConnected**(**int u**){** // O(v + e)

Stack**.**push**(**u**);**

inStack**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

low**[**u**]** **=** num**[**u**]** **=** vertexCount**++;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++){**

**if(**num**[\***v**]** **==** 0**)** StronglyConnected**(\***v**);**

**if(**inStack**[\***v**])** low**[**u**]** **=** min**(**low**[\***v**],** low**[**u**]);**

**}**

**if(**low**[**u**]** **==** num**[**u**]){**

vector**<**int**>** newSCC**;**

**while(!**Stack**.**empty**()){**

int v **=** Stack**.**top**();**

newSCC**.**push\_back**(**v**);**

inStack**[**v**]** **=** **false;**

Stack**.**pop**();**

**if(**v **==** u**)** **break;**

**}**

SCC**.**push\_back**(**newSCC**);**

**}**

**}**

***Lowest Common Ancestor***

// NOTE: Vertexes indexed from 0 to v - 1.

// IMPORTANT: parent[root] must be root.

#include <algorithm>

**using** **namespace** std**;**

#define MAXV 100000

#define LOGV 20

int level**[**MAXV**];** // Level of vertex u from root.

int parent**[**MAXV**];** // Direct parent of vertex u.

int P**[**MAXV**][**LOGV**];**

int logarithm**(**int v**){** // O(log v)

**for(**int i **=** 0**;** **true;** i**++)**

**if(**v **<=** 1 **<<** i**)** **return** i**;**

**}**

void PreprocessDP**(**int v**){** // O(v log v)

int log **=** logarithm**(**v**);**

**for(**int i **=** 0**;** i **<** v**;** i**++)**

P**[**i**][**0**]** **=** parent**[**i**];**

**for(**int j **=** 1**;** j **<=** log**;** j**++)**

**for(**int i **=** 0**;** i **<** v**;** i**++)**

P**[**i**][**j**]** **=** P**[**P**[**i**][**j **-** 1**]][**j **-** 1**];**

**}**

int LCA**(**int v**,** int a**,** int b**){** // O(log v)

int log **=** logarithm**(**v**);**

**if(**level**[**a**]** **<** level**[**b**])** swap**(**a**,** b**);**

**for(**int i **=** log**;** i **>=** 0**;** i**--)**

**if(**level**[**a**]** **-** **(**1 **<<** i**)** **>=** level**[**b**])**

a **=** P**[**a**][**i**];**

**if(**a **==** b**)** **return** a**;**

**for(**int i **=** log**;** i **>=** 0**;** i**--)**

**if(**P**[**a**][**i**]** **!=** P**[**b**][**i**])**

a **=** P**[**a**][**i**],** b **=** P**[**b**][**i**];**

**return** P**[**a**][**0**];**

**}**

***Maximum Cardinality Bipartite Matching***

// NOTE: Number of vertexes v = L + R.

// Left vertexes indexed from 0 to L - 1.

// Right vertexes indexed from L to v - 1.

// Matching can be reconstructed from array match.

#include <vector>

#include <algorithm>

**using** **namespace** std**;**

#define MAXV 1000

int match**[**MAXV**];**

bool visited**[**MAXV**];**

vector**<**int**>** E**[**MAXV**];**

int AlternatingPath**(**int u**){** // O(v + e)

**if(**u **==** **-**1**)** **return** 1**;**

**if(**visited**[**u**])** **return** 0**;**

visited**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++){**

**if(**AlternatingPath**(**match**[\***v**])){**

match**[\***v**]** **=** u**;**

**return** 1**;**

**}**

**}**

**return** 0**;**

**}**

int MaximumBipartiteMatching**(**int L**,** int R**){**

int matching **=** 0**;**

fill**(**match**,** match **+** L **+** R**,** **-**1**);**

**for(**int u **=** 0**;** u **<** L**;** u**++){**

fill**(**visited**,** visited **+** L**,** **false);**

matching **+=** AlternatingPath**(**u**);**

**}**

**return** matching**;**

**}** // O(v^2 + ve)

***Maximum Flow***

// Edmonds Karp implementation using BFS.

// NOTE: Vertexes indexed from 0 to n - 1.

// Remember to reset edges and capacity values.

// Complexity O(ve^2) but much better on practice.

#include <queue>

#include <vector>

#include <climits>

**using** **namespace** std**;**

#define MAXV 100

#define INF INT\_MAX

int dist**[**MAXV**];**

int parent**[**MAXV**];**

int capacity**[**MAXV**][**MAXV**];**

vector**<**int**>** E**[**MAXV**];**

int AugmentingPath**(**int v**,** int source**,** int sink**){**

fill**(**dist**,** dist **+** v**,** INF**);**

fill**(**parent**,** parent **+** v**,** **-**1**);**

queue**<**int**>** bfs**;**

bfs**.**push**(**source**);**

dist**[**source**]** **=** 0**;**

vector**<**int**>::**iterator w**;**

**while(!**bfs**.**empty**()){**

int u **=** bfs**.**front**();**

bfs**.**pop**();**

**if(**u **==** sink**)** **break;**

**for(**w **=** E**[**u**].**begin**();** w **!=** E**[**u**].**end**();** w**++){**

**if(**capacity**[**u**][\***w**]** **>** 0 **&&** dist**[\***w**]** **==** INF**){**

dist**[\***w**]** **=** dist**[**u**]** **+** 1**;**

parent**[\***w**]** **=** u**;**

bfs**.**push**(\***w**);**

**}**

**}**

**}**

int u **=** sink**;**

int flow **=** INF**;**

**while(**parent**[**u**]** **!=** **-**1**){**

flow **=** min**(**capacity**[**parent**[**u**]][**u**],** flow**);**

u **=** parent**[**u**];**

**}**

**if(**flow **==** INF**)**

**return** 0**;**

u **=** sink**;**

**while(**parent**[**u**]** **!=** **-**1**){**

capacity**[**parent**[**u**]][**u**]** **-=** flow**;**

capacity**[**u**][**parent**[**u**]]** **+=** flow**;**

u **=** parent**[**u**];**

**}**

**return** flow**;**

**}** // O(e)

int MaximumFlow**(**int v**,** int source**,** int sink**){**

int maxFlow **=** 0**;**

int augmentFlow **=** 1**;**

**while(**augmentFlow **>** 0**){** // O(ve)

augmentFlow **=** AugmentingPath**(**v**,** source**,** sink**);**

maxFlow **+=** augmentFlow**;**

**}**

**return** maxFlow**;**

**}** // O(ve^2)

***Minimum Cut***

// NOTE: Vertexes indexed from 0 to n - 1.

// Maximum Flow required before using Minimum Cut.

// Remember to define value of UNDIRECTED properly.

#include <vector>

**using** **namespace** std**;**

#define MAXV 100

#define UNDIRECTED true

vector**<** pair**<**int**,** int**>** **>** MinCut**;** // Result

bool inSetS**[**MAXV**];**

void ReachableFromSource**(**int u**){** // O(v)

inSetS**[**u**]** **=** **true;**

vector**<**int**>::**iterator v**;**

**for(**v **=** E**[**u**].**begin**();** v **!=** E**[**u**].**end**();** v**++)**

**if(**capacity**[**u**][\***v**]** **&&** **!**inSetS**[\***v**])**

ReachableFromSource**(\***v**);**

**}**

void MinimumCut**(**int v**,** int source**){** // O(v + e)

MinCut**.**clear**();**

fill**(**inSetS**,** inSetS **+** v**,** **false);**

ReachableFromSource**(**source**);**

vector**<**int**>::**iterator w**;**

**for(**int u **=** 0**;** u **<** v**;** u**++){**

**for(**w **=** E**[**u**].**begin**();** w **!=** E**[**u**].**end**();** w**++){**

**if(**UNDIRECTED **&&** **\***w **<** u**)** **continue;**

**if(**inSetS**[**u**]** **==** inSetS**[\***w**])** **continue;**

MinCut**.**push\_back**(**make\_pair**(**u**,** **\***w**));**

**}**

**}**

**}**

**String Algorithms**

***Suffix Array***

// String must be indexed from 0 to n and

// finished by an '$' symbol or similar.

#include <stdio.h>

#include <string.h>

#include <algorithm>

**using** **namespace** std**;**

#define MAXN 100000

int SA**[**MAXN**];**

int rank**[**MAXN**];**

int tray**[**MAXN**];**

int tempSA**[**MAXN**];**

int tempRank**[**MAXN**];**

void CountingSort**(**int n**,** int k**){** // O(n)

int ranks **=** max**(**n**,** 256**);**

fill**(**tray**,** tray **+** ranks**,** 0**);**

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

**if(**n **<=** i **+** k**)** tray**[**0**]++;**

**else** tray**[**rank**[**i **+** k**]]++;**

**for(**int i **=** 0**,** j **=** 0**;** i **<** ranks**;** i**++)**

j **+=** tray**[**i**],** tray**[**i**]** **=** j **-** tray**[**i**];**

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

**if(**n **<=** SA**[**i**]** **+** k**)** tempSA**[**tray**[**0**]++]** **=** SA**[**i**];**

**else** tempSA**[**tray**[**rank**[**SA**[**i**]** **+** k**]]++]** **=** SA**[**i**];**

copy**(**tempSA**,** tempSA **+** n**,** SA**);**

**}**

void SuffixArray**(**char**\*** s**,** int n**){** // O(n log n)

copy**(**s**,** s **+** n**,** rank**);**

**for(**int i **=** 0**;** i **<** n**;** i**++)** SA**[**i**]** **=** i**;**

**for(**int i **=** 1**;** i **<** n**;** i **<<=** 1**){**

CountingSort**(**n**,** i**);**

CountingSort**(**n**,** 0**);**

int ranks **=** 0**;**

tempRank**[**SA**[**0**]]** **=** 0**;**

**for(**int j **=** 1**;** j **<** n**;** j**++){**

**if(**rank**[**SA**[**j**]]** **!=** rank**[**SA**[**j **-** 1**]]** **||**

rank**[**SA**[**j**]+**i**]** **!=** rank**[**SA**[**j**-**1**]** **+** i**])**ranks**++;**

tempRank**[**SA**[**j**]]** **=** ranks**;**

**}**

copy**(**tempRank**,** tempRank **+** n**,** rank**);**

**if(**rank**[**SA**[**n **-** 1**]]** **==** n **-** 1**)** **break;**

**}**

**}**

// Starts code for Longest Common Prefix

int LCP**[**MAXN**];**

int prev**[**MAXN**];**

int plcp**[**MAXN**];**

void LongestCommonPrefix**(**char**\*** s**,** int n**){** // O(n)

prev**[**SA**[**0**]]** **=** **-**1**;**

**for(**int i **=** 1**;** i **<** n**;** i**++)**

prev**[**SA**[**i**]]** **=** SA**[**i **-** 1**];**

int L **=** 0**;**

plcp**[**SA**[**0**]]** **=** 0**;**

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

**if(**prev**[**i**]** **==** **-**1**)** **continue;**

**while(**prev**[**i**]** **+** L **<** n **&&** i **+** L **<** n

**&&** s**[**i **+** L**]** **==** s**[**prev**[**i**]** **+** L**])** L**++;**

plcp**[**i**]** **=** L**,** L **=** max**(**L **-** 1**,** 0**);**

**}**

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

LCP**[**i**]** **=** plcp**[**SA**[**i**]];**

**}**

***Knuth-Morris-Pratt***

// Before using Text Search call Pattern Fail.

// By default, Text Search does nothing, modify

// code tagged with "Matching" and return value

// to handle the way of processing matches.

#define MAXT 1000000

#define MAXP 1000000

char T**[**MAXT**];** // Text.

char P**[**MAXP**];** // Pattern.

int F**[**MAXP**];**

void PatternFail**(**int p**){** // O(p)

F**[**0**]** **=** **-**1**;** int j **=** **-**1**;**

**for(**int i **=** 0**;** i **<** p**;** i**++){**

**while(**P**[**i**]** **!=** P**[**j**]** **&&** j **>=** 0**)** j **=** F**[**j**];**

F**[**i **+** 1**]** **=** **++**j**;**

**}**

**}**

void TextSearch**(**int t**,** int p**){** // O(t + p)

**for(**int i **=** 0**,** j **=** 0**;** i **<** t**;** i**++){**

**while(**T**[**i**]** **!=** P**[**j**]** **&&** j **>=** 0**)** j **=** F**[**j**];**

**if(++**j **==** p**){**

// Matching

j **=** F**[**j**];**

**}**

**}**

**}**

**Computational Geometry**

***Points and Vectors***

#include <cmath>

#define PI M\_PI

#define ERROR 1e-12

#define RAD (PI / 180)

bool equal**(**double a**,** double b**){** // O(1)

**return** fabs**(**a **-** b**)** **<** ERROR**;**

**}**

// Functions to manipulate points and vectors

// in a 2D euclidean space, many functions may

// change if another dimension is required.

struct Point**{**

double x**,** y**;**

bool **operator** **<(**Point cmp**)** const**{**

**if(**equal**(**x**,** cmp**.**x**))**

**return** y **<** cmp**.**y**;**

**return** x **<** cmp**.**x**;**

**}**

**};**

Point P**(**double x**,** double y**){** // O(1)

Point**\*** p **=** **new** Point**;**

p**->**x **=** x**,** p**->**y **=** y**;**

**return** **\***p**;**

**}**

// Vectors represented as points.

Point Vector**(**Point o**,** Point p**){** // O(1)

**return** P**(**p**.**x **-** o**.**x**,** p**.**y **-** o**.**y**);**

**}**

Point Scale**(**Point p**,** double k**){** // O(1)

**return** P**(**p**.**x **\*** k**,** p**.**y **\*** k**);**

**}**

// Rotate a point around origin.

Point Rotate**(**Point p**,** double angle**){** // O(1)

double rad **=** angle **\*** RAD**;**

double x **=** cos**(**rad**)** **\*** p**.**x **-** sin**(**rad**)** **\*** p**.**y**;**

double y **=** sin**(**rad**)** **\*** p**.**x **+** cos**(**rad**)** **\*** p**.**y**;**

**return** P**(**x**,** y**);**

**}**

Point Translate**(**Point p**,** Point v**){** // O(1)

**return** P**(**p**.**x **+** v**.**x**,** p**.**y **+** v**.**y**);**

**}**

double Dot**(**Point u**,** Point v**){** // O(1)

**return** u**.**x **\*** v**.**x **+** u**.**y **\*** v**.**y**;**

**}**

double Cross**(**Point u**,** Point v**){** // O(1)

**return** u**.**x **\*** v**.**y **-** v**.**x **\*** u**.**y**;**

**}**

double Magnitude**(**Point v**){** // O(1)

**return** sqrt**(**Dot**(**v**,** v**));**

**}**

// Euclidean distance between point p and q.

double Distance**(**Point p**,** Point q**){** // O(1)

**return** hypot**(**p**.**x **-** q**.**x**,** p**.**y **-** q**.**y**);**

**}**

// Angle between two vectors.

double Angle**(**Point u**,** Point v**){** // O(1)

double U **=** Magnitude**(**u**);**

double V **=** Magnitude**(**v**);**

**if(**U **==** 0 **||** V **==** 0**)** **return** 0**;**

**return** acos**(**Dot**(**u**,** v**)** **/** **(**U **\*** V**))** **/** RAD**;**

**}**

// Angle POQ between three points.

double Angle**(**Point o**,** Point p**,** Point q**){** // O(1)

**return** Angle**(**Vector**(**o**,** p**),** Vector**(**o**,** q**));**

**}**

***Lines and Segments***

// Line structure consists in two points p and q

// and three doubles a, b and c for the line equation.

// Segments represented as lines, when using segments,

// points p and q are meant to be the edges and the

// values for line equation may not be necessary.

struct Line**{**

Point p**,** q**;**

double a**,** b**,** c**;**

**};**

Line L**(**Point p**,** Point q**){** // O(1)

**if(**q **<** p**)** **return** L**(**q**,** p**);**

Line**\*** l **=** **new** Line**();**

l**->**p **=** p**,** l**->**q **=** q**;**

l**->**c **=** Cross**(**p**,** q**);**

l**->**a **=** p**.**y **-** q**.**y**;**

l**->**b **=** q**.**x **-** p**.**x**;**

**return** **\***l**;**

**}**

// Determine if two lines are parallel.

bool Parallel**(**Line l**,** Line m**){** // O(1)

**return** equal**(**l**.**a**,** m**.**a**)** **&&** equal**(**l**.**b**,** m**.**b**);**

**}**

// Compare if two lines are the same.

bool **operator** **==(**Line l**,** Line m**){** // O(1)

**return** Parallel**(**l**,** m**)** **&&** equal**(**l**.**c**,** m**.**c**);**

**}**

// Intersection point of two lines.

Point Intersection**(**Line l**,** Line m**){** // O(1)

double div **=** m**.**a **\*** l**.**b **-** l**.**a **\*** m**.**b**;**

double x **=** m**.**b **\*** l**.**c **-** l**.**b **\*** m**.**c**;**

double y **=** m**.**a **\*** l**.**c **-** l**.**a **\*** m**.**c**;**

**return** P**(**x **/** div**,** y **/** **-**div**);**

**}**

Point CloserPointInLine**(**Point p**,** Line l**){** // O(1)

Point u **=** Vector**(**l**.**p**,** l**.**q**),** v **=** Vector**(**l**.**p**,** p**);**

double scale **=** Dot**(**u**,** v**)** **/** pow**(**Magnitude**(**u**),** 2**);**

**return** Translate**(**l**.**p**,** Scale**(**u**,** scale**));**

**}**

Point CloserPointInSegment**(**Point p**,** Line s**){** // O(1)

Point u **=** Vector**(**s**.**p**,** s**.**q**),** v **=** Vector**(**s**.**p**,** p**);**

double scale **=** Dot**(**u**,** v**)** **/** pow**(**Magnitude**(**u**),** 2**);**

**if(**scale **<** 0**)** **return** P**(**s**.**p**.**x**,** s**.**p**.**y**);**

**if(**scale **>** 1**)** **return** P**(**s**.**q**.**x**,** s**.**q**.**y**);**

**return** Translate**(**s**.**p**,** Scale**(**u**,** scale**));**

**}**

// Minimum distance between point p and line l.

double DistanceToLine**(**Point p**,** Line l**){** // O(1)

**return** Distance**(**CloserPointInLine**(**p**,** l**),** p**);**

**}**

// Minimum distance between point p and segment s.

double DistanceToSegment**(**Point p**,** Line s**){** // O(1)

**return** Distance**(**CloserPointInSegment**(**p**,** s**),** p**);**

**}**

// Counter clockwise test, returns true if point r

// is on the left side of line pq, false otherwise.

bool CCW**(**Point p**,** Point q**,** Point r**){** // O(1)

**return** Cross**(**Vector**(**p**,** q**),** Vector**(**p**,** r**))** **>** 0**;**

**}**

bool Collinear**(**Point p**,** Point q**,** Point r**){** // O(1)

**return** equal**(**Cross**(**Vector**(**p**,** q**),** Vector**(**p**,** r**)),** 0**);**

**}**

bool PointInSegment**(**Point p**,** Line s**){**

double y1 **=** s**.**p**.**y **<** s**.**q**.**y**?** s**.**p**.**y**:** s**.**q**.**y**;**

double y2 **=** s**.**p**.**y **>** s**.**q**.**y**?** s**.**p**.**y**:** s**.**q**.**y**;**

**if(**p**.**x **<** s**.**p**.**x **||** s**.**q**.**x **<** p**.**x**)** **return** **false;**

**if(**p**.**y **<** y1 **||** y2 **<** p**.**y**)** **return** **false;**

**return** Collinear**(**p**,** s**.**p**,** s**.**q**);**

**}**

***Polygons***

// Polygons represented as a vector of points,

// the first point in vector is also the last one.

#include <vector>

**using** **namespace** std**;**

// Perimeter of polygon.

double Perimeter**(**vector**<**Point**>** P**){**

double result **=** 0**;**

**for(**int i **=** 1**;** i **<** P**.**size**();** i**++)**

result **+=** Distance**(**P**[**i **-** 1**],** P**[**i**]);**

**return** result**;**

**}**

// Area of polygon.

double Area**(**vector**<**Point**>** P**){**

double result **=** 0**;**

**for(**int i **=** 1**;** i **<** P**.**size**();** i**++)**

result **+=** Cross**(**P**[**i **-** 1**],** P**[**i**]);**

**return** fabs**(**result**)** **/** 2.0**;**

**}**

// Tests if a point p is inside of polygon P.

// If the point is on the bounds is considered inside.

bool PointInPolygon**(**Point p**,** vector**<**Point**>** P**){**

**for(**int i **=** 1**;** i **<** P**.**size**();** i**++)**

**if(**PointInSegment**(**p**,** L**(**P**[**i**],** P**[**i **-** 1**])))**

**return** **true;**

double angle **=** 0**;**

**for(**int i **=** 1**;** i **<** P**.**size**();** i**++){**

double alpha **=** Angle**(**p**,** P**[**i **-** 1**],** P**[**i**]);**

**if(**CCW**(**p**,** P**[**i**],** P**[**i **-** 1**]))**

alpha **=** **-**alpha**;**

angle **+=** alpha**;**

**}**

**if(**round**(**angle**)** **==** 360**)**

**return** **true;**

**return** **false;**

**}**

***Convex Hull***

// Receives a vector of points in the 2D plane,

// returns a polygon with the points that forms

// the convex hull sorted in counter-clockwise order.

// NOTE: Function deletes collinear points, if this

// points are required, modify CCW function.

#include <vector>

#include <algorithm>

**using** **namespace** std**;**

vector**<**Point**>** ConvexHull**(**vector**<**Point**>** P**){**

vector**<**Point**>** U**,** L**;**

sort**(**P**.**begin**(),** P**.**end**());**

**for(**int i **=** 0**;** i **<** P**.**size**();** i**++){**

**while(**L**.**size**()** **>** 1**){**

Point p **=** L**[**L**.**size**()** **-** 2**];**

Point q **=** L**[**L**.**size**()** **-** 1**];**

**if(**CCW**(**p**,** q**,** P**[**i**]))** **break;**

**else** L**.**pop\_back**();**

**}**

L**.**push\_back**(**P**[**i**]);**

**}**

**if(**L**.**size**()** **>** 1**)** L**.**pop\_back**();**

**for(**int i **=** P**.**size**()** **-** 1**;** i **>=** 0**;** i**--){**

**while(**U**.**size**()** **>** 1**){**

Point p **=** U**[**U**.**size**()** **-** 2**];**

Point q **=** U**[**U**.**size**()** **-** 1**];**

**if(**CCW**(**p**,** q**,** P**[**i**]))** **break;**

**else** U**.**pop\_back**();**

**}**

U**.**push\_back**(**P**[**i**]);**

**}**

**for(**int i **=** 0**;** i **<** U**.**size**();** i**++)**

L**.**push\_back**(**U**[**i**]);**

**return** L**;**

**}**

**Algorithm Hungariano**

**#include <cstdio>**

**#include <cstdlib>**

**#include <cstring>**

**#include <algorithm>**

**#include <queue>**

**using namespace std;**

**#define N 55 //max number of vertices in one part**

**#define INF 100000000 //just infinity**

**int cost[N][N]; //cost matrix; fill unconnected with -INF; for minimum, negate weights**

**int n, matched; //n workers and n jobs**

**int lx[N], ly[N]; //labels of X and Y parts**

**int xy[N]; //xy[x] - vertex that is matched with x,**

**int yx[N]; //yx[y] - vertex that is matched with y**

**bool S[N], T[N]; //sets S and T in algorithm**

**int slack[N]; //as in the algorithm description**

**int slackx[N]; //slackx[y] such a vertex, that l(slackx[y]) + l(y) - w(slackx[y],y) = slack[y]**

**int p[N]; //array for memorizing alternating paths**

**void init() {**

**matched = 0;**

**fill\_n(xy, n, -1);**

**fill\_n(yx, n, -1);**

**fill\_n(lx, n, 0);**

**fill\_n(ly, n, 0);**

**for (int x = 0; x < n; x++)**

**lx[x] = \*max\_element(cost[x], cost[x] + n);**

**}**

**void update\_labels() {**

**int x, y, delta = INF;**

**for (y = 0; y < n; y++) //calculate delta using slack**

**if (!T[y]) delta = min(delta, slack[y]);**

**for (x = 0; x < n; x++) //update X labels**

**if (S[x]) lx[x] -= delta;**

**for (y = 0; y < n; y++) //update Y labels**

**if (T[y]) ly[y] += delta;**

**for (y = 0; y < n; y++) //update slack array**

**if (!T[y]) slack[y] -= delta;**

**}**

**//x - current vertex, px - vertex from X before x in the alternating path,**

**//so we add edges (px, xy[x]), (xy[x], x)**

**void add\_to\_tree(int x, int px) {**

**S[x] = true; //add x to S**

**p[x] = px; //we need this when augmenting**

**for (int y = 0; y < n; y++) //update slacks, because we add new vertex to S**

**if (lx[x] + ly[y] - cost[x][y] < slack[y]) {**

**slack[y] = lx[x] + ly[y] - cost[x][y];**

**slackx[y] = x;**

**}**

**}**

**//main function of the algorithm**

**void augment() {**

**int x, y, root; //just counters and root vertex**

**queue<int> q; //queue for bfs**

**while (matched != n) { //check whether matching is already perfect**

**q.empty();**

**fill\_n(S, n, false); //init set S**

**fill\_n(T, n, false); //init set T**

**fill\_n(p, n, -1); //init set p - for the alternating tree**

**for (x = 0; x < n; x++) //finding root of the tree**

**if (xy[x] == -1) {**

**root = x;**

**q.push(x);**

**p[x] = -2;**

**S[x] = true;**

**break;**

**}**

**for (y = 0; y < n; y++) { //initializing slack array**

**slack[y] = lx[root] + ly[y] - cost[root][y];**

**slackx[y] = root;**

**}**

**while (true) { //main cycle**

**while (q.size()) { //building tree with bfs cycle**

**x = q.front(); q.pop(); //current vertex from X part**

**for (y = 0; y < n; y++) //iterate through all edges in equality graph**

**if (cost[x][y] == lx[x] + ly[y] && !T[y]) {**

**if (yx[y] == -1) break; //an exposed vertex in Y found, so augmenting path exists!**

**T[y] = true; //else just add y to T,**

**q.push(yx[y]); //add vertex yx[y], which is matched with y, to the queue**

**add\_to\_tree(yx[y], x); //add edges (x,y) and (y,yx[y]) to the tree**

**}**

**if (y < n) break; //augmenting path found!**

**}**

**if (y < n) break; //augmenting path found!**

**update\_labels(); //augmenting path not found, so improve labelling**

**//in this cycle we add edges that were added to the equality graph as a**

**//result of improving the labelling, we add edge (slackx[y], y) to the tree if**

**//and only if !T[y] && slack[y] == 0, also with this edge we add another one**

**//(y, yx[y]) or augment the matching, if y was exposed**

**for (y = 0; y < n; y++)**

**if (!T[y] && slack[y] == 0) {**

**if (yx[y] == -1) { //exposed vertex in Y found - augmenting path exists!**

**x = slackx[y];**

**break;**

**} else {**

**T[y] = true; //else just add y to T,**

**if (!S[yx[y]]) {**

**q.push(yx[y]); //add vertex yx[y], which is matched with y, to the queue**

**add\_to\_tree(yx[y], slackx[y]); // add edges (x,y) and (y, yx[y]) to the tree**

**}**

**}**

**}**

**if (y < n) break; //augmenting path found!**

**}**

**matched++; //augmenting path found, so increment matching**

**//in this cycle we inverse edges along augmenting path**

**for (int cx = x, cy = y, ty; cx != -2; cx = p[cx], cy = ty) {**

**ty = xy[cx];**

**yx[cy] = cx;**

**xy[cx] = cy;**

**}**

**} //go to step 1 of the algorithm**

**}**

**int hungarian() {**

**int ret = 0; //weight of the optimal matching**

**init();**

**augment();**

**for (int x = 0; x < n; x++) //forming answer there**

**ret += cost[x][xy[x]];**

**return ret;**

**}**

**int main() {**

**n = 3;**

**cost[0][0] = 5;**

**cost[0][1] = 0;**

**cost[0][2] = 20;**

**cost[1][0] = 0;**

**cost[1][1] = 5;**

**cost[1][2] = 0;**

**cost[2][0] = 0;**

**cost[2][1] = 0;**

**cost[2][2] = 5;**

**printf("%d\n", hungarian());**

**getchar();**

**}**

**Extendido de Euclides**

**// finds sa + tb = gcd(a, b) = r**

**int extended\_gcd(int a, int b) {**

**int s = 1, ns = 0, t = 0, nt = 1, r = b, nr = a, p, q;**

**while (nr) {**

**q = r / nr;**

**p = ns;**

**ns = s - q \* p;**

**s = p;**

**p = nt;**

**nt = t - q \* p;**

**t = p;**

**p = nr;**

**nr = r - q \* p;**

**r = p;**

**}**

**return r; // gcd(a, b)**

**}**

**// finds modular inversor of a mod b**

**// x value such as: a \* x = 1 % b**

**int modinv(int a, int b) {**

**int t = 0, nt = 1, r = b, nr = a, p, q;**

**while (nr) {**

**q = r / nr;**

**p = nt;**

**nt = t - q \* p;**

**t = p;**

**p = nr;**

**nr = r - q \* p;**

**r = p;**

**}**

**if (r > 1) return -1; // a is not inversible**

**if (t < 0) t += b;**

**return t;**

**}**

**Fraccion**

**pair <int, int> Decimal\_a\_Fraccion(double x){**

**int rq = 0;**

**double min\_diff = 1e100;**

**double l = x;**

**for (int q = 1; q <= 100000; q++){**

**double pr = l \* q;**

**int val = (int)(pr + 0.5);**

**double diff = fabs(pr - val) / q;**

**if (diff < min\_diff)**

**min\_diff = diff, rq = q;**

**}**

**int rp = (int)(l \* rq + 0.5);**

**int g = \_\_gcd(rp, rq);**

**return make\_pair(rp / g, rq / g);**

**}**

**HeavyLigth Descompochichon**

**#include <stdio.h>**

**#include <map>**

**#include <vector>**

**#include <string>**

**#include <algorithm>**

**#define MAXV 300002**

**using namespace std;**

**int size[MAXV];**

**int parent[MAXV];**

**vector<int> E[MAXV];**

**vector<vector<int> > HL; // Compressed tree.**

**pair<int, int> hl\_parent[MAXV]; // Parents in HL.**

**pair<int, int> component[MAXV]; // Component in HL.**

**int CalculateSize(int u, int p) {**

**size[u] = 0;**

**parent[u] = p;**

**for (int v = 0; v < E[u].size(); ++v) {**

**if (E[u][v] == p) continue;**

**size[u] += CalculateSize(E[u][v], u);**

**}**

**// Look for heavy components.**

**component[u].first = -1;**

**for (int v = 0; v < E[u].size(); ++v) {**

**if (E[u][v] == p) continue;**

**// If there is a heavy edge.**

**if (size[E[u][v]] > size[u] / 2) {**

**component[u].first = component[E[u][v]].first;**

**HL[component[u].first].push\_back(u);**

**}**

**}**

**// Doesn't belong to any heavy component.**

**if (component[u].first == -1) {**

**vector<int> heavy;**

**heavy.push\_back(u);**

**component[u].first = HL.size();**

**HL.push\_back(heavy);**

**}**

**return ++size[u];**

**}**

**void BuildHeavyLight(int root) {**

**HL.clear();**

**CalculateSize(root, -1);**

**for (int i = 0; i < HL.size(); ++i) {**

**reverse(HL[i].begin(), HL[i].end());**

**for(int j = 0; j < HL[i].size(); ++j)**

**component[HL[i][j]].second = j;**

**int p = parent[HL[i][0]];**

**int comp = (p == -1)? -1: component[p].first;**

**hl\_parent[i] = make\_pair(comp, p);**

**}**

**}**

**// PROBLEM SPECIFIC**

**char input[30];**

**map<string, string> values[MAXV];**

**map<string, vector<int> > M[MAXV];**

**int main() {**

**int n, p, k, r;**

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

**for (int i = 1; i <= n; ++i) {**

**scanf("%d%d", &p, &k);**

**if (p != 0) {**

**E[i].push\_back(p);**

**E[p].push\_back(i);**

**} else r = i;**

**for (int j = 0; j < k; ++j) {**

**scanf("%s", input);**

**for(p = 0; input[p] != '='; ++p) {}**

**string key(input, input + p), value(input + p + 1);**

**values[i][key] = value;**

**}**

**}**

**BuildHeavyLight(r);**

**for (int i = 0; i < HL.size(); ++i) {**

**for (int j = 0; j < HL[i].size(); ++j) {**

**map<string, string>::iterator it;**

**for (it = values[HL[i][j]].begin(); it != values[HL[i][j]].end(); ++it)**

**M[i][it->first].push\_back(j);**

**}**

**}**

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

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

**scanf("%d%s", &p, input);**

**string key(input);**

**while (p != -1) {**

**if (M[component[p].first].count(key)) {**

**vector<int> v = M[component[p].first][key];**

**int w = upper\_bound(v.begin(), v.end(), component[p].second) - v.begin();**

**if (w > 0) { printf("%s\n", values[HL[component[p].first][v[w - 1]]][key].c\_str()); break; }**

**}**

**p = hl\_parent[component[p].first].second;**

**}**

**if(p == -1)**

**puts("N/A");**

**fflush(stdout);**

**}**

**return 0;**

**}**