Changes in version 2.

Alternate solution provided for problem B.

Removed the DP from problem G as it can be solved greedily.

Linked in an extra tutorial for problem H.

Solution description and implementation added for problem L.

Problem A
Secure PINs

Time Limit: 2 seconds

Most people are aware of the importance of selecting a good password for their computer and e-mail accounts. However, they pay little attention when they choose their credit card and bank card PINs, even though they can probably unlock a lot of wealth.

Your task is to write a program to assess the security level of a collection of PINs. A PIN that contains the same digit three times or a sequence of three consecutive digits (such as 345 and 654) is to be assessed as weak. Otherwise, it is considered acceptable. Note that sequences that wrap around, like “391312” and “098165”, are not considered consecutive.

Input

The input starts with an integer N, on a line by itself, that represents the number of test cases. $1 \leq N \leq 1000$. The description for each test case consists of a six-digit non-negative number on a line by itself.

Output

The output consists of a single line, for each test case, which contains your program's assessment of the PIN as “WEAK” or “ACCEPTABLE”.

Sample Input	Output for the Sample Input
9	ACCEPTABLE
024578	ACCEPTABLE
248905	ACCEPTABLE
391312	ACCEPTABLE
098245	WEAK
145698	WEAK
212324	WEAK
986541	ACCEPTABLE
968541	ACCEPTABLE
540872	

A: Secure PINs

The easiest problem of the contest. The only possible problems that I can see is going off the end of the string, or maybe thinking that if numbers go ”890” that this is considered weak, but then this ambiguity is handled in the sample test data in the second case.

I would say the lesson to learn from this problem, is that the sample test data can often be useful in resolving ambiguities in the problem description.

The time complexity of this solution is $\mathcal{O}(N)$, it is linear in the input.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

int main(){
  int ttt=getint();
  for(int tt=0; tt<ttt; tt++){
    string s;
    cin >> s;
    int ok = 1;
    // check ascending / descending
    for(int i=0; i<sz(s)-2; i++)
      if(
          (s[i]==s[i+1]-1 && s[i+1]==s[i+2]-1) ||
          (s[i]==s[i+1]+1 && s[i+1]==s[i+2]+1)
      )
        ok = 0;
    // check no 3 of the same digit
    int cnt[10] = {0};
    for(int i=0; i<sz(s); ++i)
      cnt[s[i]-'0']++;
    for(int i=0; i<10; ++i)
      if(cnt[i] >= 3)
        ok = 0;
    // output
    printf(ok ? "ACCEPTABLE\n" : "WEAK\n");
  }
  return 0;
}
```

Problem B
The Large Family

Time Limit: 2 seconds

The "Occupy the Kitchen" movement by the **five** children of the Large family, has taken Mr. and Mrs. Large by surprise. The children raised banners demanding the equal distribution of mangos after a large number of mangos went missing from the fridge overnight. They presented their demands as the family food constitution.

The Large Family Food Constitution

- A mango may only be divided into two halves.
- Children shall receive equal amounts of mangos.
- Mum and Dad shall receive equal amounts of mangos.
- Allocation may differ by no more than half a mango amongst all family members.
- Each family member must receive the maximum amount possible without violating the above constitutional items.
- Extra amounts of mangos that result in the violation of the above constitutional items shall go to the goat. Mangos going to the goat must be kept to a minimum.

Mr. and Mrs Large have agreed to the constitution on the condition that software is developed to compute the correct allocation of mangos. Your task is to write a program that allocates mangos to each family member and the goat.

Input

The input consists of many test cases. The description for each test case consists of an integer, $0 < G \leq 300$, on a line by itself that represents the total number of mangos to be divided.

A line with a single zero indicates the end of data and should not be processed.

Output

The output consists of a single line, for each test case, which contains four values separated by single spaces. The first is the number of mangos for the goat, the second is the number of mangos for dad, the third is the number of mangos for mum and the fourth is the number of mangos for each child. Each value is written as a decimal with a single digit after the decimal point.

Sample Input	Output for the Sample Input
41 0	0.0 5.5 5.5 6.0

B: The Large Family

Something to really get used to, is staying in the integers whenever possible. So we will count the number of halves of mango as opposed to the number of mangos.

Since both parents receive the same number of halves, and so do all 5 children, we simply need to find np and nc, the number of halves per parent and child respectively, such that $n - 2 * np - 5 * nc$ is minimal, where n is the number of halves of mango available.

The constraints are small enough that you can brute force over all possible assignments and just keep the one which minimises the amount of mango given to the goat.

The time complexity of this solution is $\mathcal{O}(G^2)$ per test case.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

int main(){
  int n;
  while( (n=getint()) ){
    n *= 2; // there are 2n halves
    int np, nc, ng=n; // record the best values found
    for(int i=0; 2*i<=n; ++i) // brute force np
      for(int j=0; 5*j<=n; ++j) // brute force ng
        if(
            abs(i-j)<=1 // allocation differs by at most half amongst all family members
            && 2*i + 5*j <= n // amount distributed must be <= amount available
            && n - 2*i - 5*j < ng // minimise the amount given to the goat
        ){
          np = i;
          nc = j;
          ng = n - 2*np - 5*nc;
        }
    printf("%d.%d %d.%d %d.%d %d.%d\n",
        ng/2, 5*(ng%2),
        np/2, 5*(np%2),
        np/2, 5*(np%2),
        nc/2, 5*(nc%2)
    );
  }
  return 0;
}
```

Alternatively, it is possible to solve each test case in $\mathcal{O}(1)$. Here I describe the solution presented by Rupert Lester in the SPPContest facebook group.

Since allocations differ by at most half amongst all family members. Then there is some x such that (np, nc) is either (x, x), $(x + 1, x)$, or $(x, x + 1)$. And this x is $n/7$ (again n is the number of halves of mango). Figure out how much the goat would get if $(np, nc) = (x, x)$, this is just $n\%7$. And if you can take 5 off the goat, give it to the kids, else if you can take 2 of the goat, give it to the parents.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

int main(){
  int n, np, nc, ng;
  while( (n=getint()) ){
    n *= 2; // there are 2n halves
    np = nc = n / 7;
    ng = n % 7;
    if(ng >= 5){
      ++nc;
      ng -= 5;
    }else if(ng >= 2){
      ++np;
      ng -= 2;
    }
    printf("%.1lf %.1lf %.1lf %.1lf\n", ng/2.0, np/2.0, np/2.0, nc/2.0);
  }
  return 0;
}
```

Problem C
Start-up Network

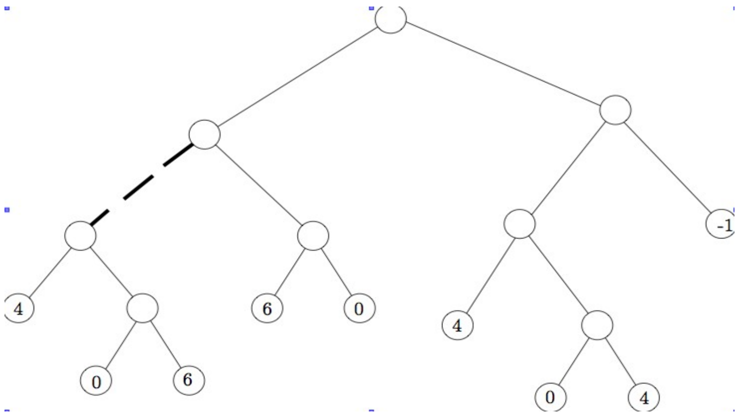
Time Limit: 15 seconds

A start-up communication company plans to build a proprietary network using cheaply available 3-way routers[1] from a local supplier. The network will be configured in a tree-like architecture with all the customers as its leaf nodes.

Customers may be engaged in one-to-one calls or in conference calls but a customer cannot be simultaneously engaged in more than one call. An active one-to-one call uses the links on the unique path between the two leaf nodes corresponding to the two customers. An active conference call uses the links of the connected sub-tree that spans leaf nodes corresponding to the participating customers.

The company is worried about the negative effect that too many active calls using a single link may have on the overall performance of its network. The company wants you to calculate the maximum number of active calls using any link (that is, any edge of the tree) in the network for a given set of calls.

The dashed link in the example network, shown below,



is used by the three active calls, which is the maximum value.

1 A *router* is a device that forwards data packets between two, or more, networks. A router is said to be 3-way if it only has three communication ports.

Input

The input consists of a number of test cases, where the data for each case is a full binary rooted tree[2] of T leaf nodes on a line by itself. A tree is described by a set of matched parentheses and a set of integers in the range of “-1” to “T-1”, $3 \leq T \leq 5000$. Parentheses and integers are separated by single spaces.

A pair of matching parentheses represents the root of one full binary rooted tree and an integer represents a leaf node in the tree. Leaf nodes with the same integer value correspond to the subset of customers engaged in an active call. A leaf node with a value of “-1” indicates a customer who is not engaged in a call.

Two matched parentheses, separated by a single space, on a line by themselves indicate the end of input data and should not be processed.

Output

For each test case, the output consists of a single line. The line begins with the word “Tree” followed by a space, an integer N, the character sequence “:␣”, and then an integer indicating the maximum communication load on any tree edge. N is the number of the test case starting with the value “1”, and the symbol “␣” indicates a single space.

Sample Input	Output for the Sample Input
((0 1) ((1 1) 0)) ((0 (0 3)) (((1 0) (1 0)) 3)) (((4 (0 6)) (6 0)) ((4 (0 4)) -1)) ( )	Tree 1: 2 Tree 2: 2 Tree 3: 3

2 A *full binary rooted tree* consists of a single leaf node , or a root node connected to two full binary rooted trees. In such a tree, each node has either zero or two child nodes.

C: Start-up Network

The problem is: Given a binary tree with some leaves coloured. For all pairs of leaves with the same colour, colour the path between them that colour. What is the maximal amount of different colours on a single edge.

The only observation to make is that for a set of leaves of one colour, it is equivalent to just colour from each of them to their lowest common ancestor (lca). With the constraints so low ($T = 5000$) an $\mathcal{O}(T^2)$ solution which just walks from each leaf to the lca colouring the edges will work.

When colouring the edges, if two leaves of the same colour traverse the same edge, be sure to only count the colour once.

Lowest common ancestor is something which may not be familiar to many of you, and so I direct you to the following tutorials:

`http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor`

`http://e-maxx.ru/algo/lca_simpler`

The $O(n \lg n)$ preprocessing and $O(\lg n)$ query solution to the lowest common ancestor problem should be all that you ever need for informatics competitions and is very simple to implement once you understand it. The idea behind it is that node x's parent $2i$ levels up, is the parent i levels up from the node i levels up from x.

The time complexity of this solution is $\mathcal{O}(T^2)$ per test case.

The parsing solution here is inspired by the recursive parsing solution presented by George Wilson in the SPPContest facebook group. Thank you for sharing your appreach George, it is much cleaner than the one that I used during the contest.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

// the graph
const int N = 14;
vector< vector< int > > a;
int colour[1<<N], upto;
int lca[N+1][1<<N]; // lca[i][j] is the node you get if you step up 2^i levels from j
int height[1<<N];

// find everybodys parent and height in the tree
void initLca(int i, int p, int h){
  height[i] = h;
  lca[0][i] = p;
  for(int j=0; j<sz(a[i]); ++j)
    if(a[i][j] != p)
      initLca(a[i][j], i, h+1);
}

// p is the id of my parent
```

```
void parse(int l = 0){
  // my id number in the graph
  int id = upto++;
  // read in both children
  char c;
  for(int i=0; i<2; ++i){
    c = cin.peek();
    if(c == '-' || (c>='0' && c<='9')){
      a.push_back(vector<int>());
      a[id].push_back(upto);
      a[upto].push_back(id);
      scanf("%d ", &colour[upto++]);
    }else if(c == '('){
      scanf("( ");
      a.push_back(vector<int>());
      a[id].push_back(upto);
      a[upto].push_back(id);
      parse(l+1);
    }else{ // c == ')', this only happens on the very final case
      break;
    }
  }
  scanf(") ");
}

int findLca(int p, int q){

  // let p be lower in the tree than q, ie greater depth
  if(height[p] < height[q]) swap(p,q);

  // find ancestor of p on same level as q
  // ie, whilst can jump up without jumping higher than q, jump
  for(int i=N; i>=0; --i)
    if(height[p]-(1<<i) >= height[q])
      p = lca[i][p];

  // must handle this base case, it is not handled by following code
  if(p==q) return p;

  // compute LCA(p,q) using the values in precomputed P
  // ie, whilst both can jump up without meeting, do so - result is one below answer
  for(int i=N; i>=0; --i)
    if(lca[i][p] != lca[i][q]){
      p = lca[i][p];
      q = lca[i][q];
    }

  // return one above where we got to - it is the LCA
  return lca[0][p];

}

int main(){
  int tt = 0;
  while(true){

    // clear data between cases
    a = vector< vector< int > >(1);
    memset(colour, -1, sizeof(colour));
    memset(lca, 0, sizeof(lca));
    memset(height, 0, sizeof(height));
    upto = 0;

    // turn the input into a tree in a more user friendly format
    scanf(" ( "); // throw this away, parse assumes the current node has already been opened up
    parse();

    if(upto <= 2)
      break;

    initLca(0, -1, 0);

    // construct the lca array, given that level 0 is already in place
    // lca array is very easy to construct, and then gives you log time lca query
    for(int i=1; i<=N; ++i)
      for(int j=0; j<upto; ++j)
        lca[i][j] = lca[i-1][j]==-1 ? -1 : lca[i-1][lca[i-1][j]];

    // for each colour, find lca and colour the edges
    set<int> coloursSeen;
    vector<int> timesColoured(upto, 0); // an edge will have the same id as the vertex below it in the tree
    // traverse all vertices to find all used colours
    for(int i=0; i<upto; ++i)
      if(colour[i] != -1 && !coloursSeen.count(colour[i])){
        coloursSeen.insert(colour[i]); // only perform this once for each colour
        vector<int> seen(upto, 0);
        int lc = i;
        // find the lca
        for(int j=i+1; j<upto; ++j)
          if(colour[j] == colour[i])
            lc = findLca(lc, j);
        // walk from each person upto the lca
        for(int j=i; j<upto; ++j)
          if(colour[j] == colour[i])
            for(int k=j; k!=lc; k=lca[0][k]) // N.B. lca[0][k] is just the parent of k
              if(!seen[k]){ // Remember, for each edge, it only gets each colour once
                ++timesColoured[k];
                seen[k] = 1;
              }
      }

    // find the most coloured edge
    int ret = 0;
    for(int i=0; i<upto; ++i)
      ret = max(ret, timesColoured[i]);

    // output
    printf("Tree %d: %d\n", ++tt, ret);

  }
  return 0;
}
```

Problem D
Clock Splitter

Time Limit: 2 seconds

An analogue clock has the first twelve natural numbers placed in an equally spaced fashion in a clockwise direction on its face. It is possible to draw one line across this regular clock such that the sum of the numbers on both sides of the line are equal, as shown in figure A.

It is not possible to draw such a line for a clock with the first five natural numbers on its face. However it is possible to draw a line such that the sum of the numbers on both sides of the line differ by one, as shown in figure B.

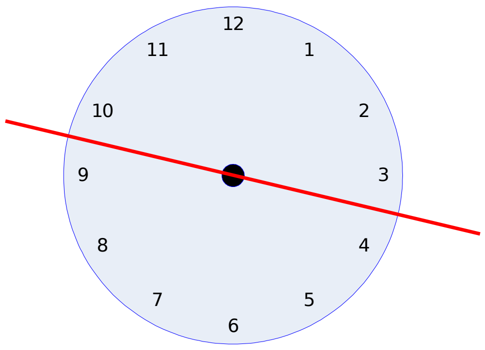


Figure A

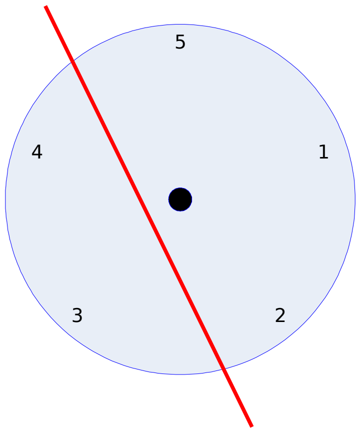


Figure B

Your task is to write a program to find where the line can be drawn for a clock with the first *N* natural numbers such that the sum of the numbers on both sides of the line are as close as possible to each other.

For some values of *N*, there will be more than one possible solution. Your program must report only the line with the smallest starting number.

Input

The input contains a number of test cases with one input value N per case, 2 ≤ N ≤ 100000 , which is the largest value on the clock face.

A value of zero (0) on a line by itself indicates the end of input and should not be processed.

Output

The output consists of a single line, for each test case, which contains two integers separated by a single space. The integers represent the smallest number followed by the largest number, in the clockwise direction, on the same side of the splitting line.

Sample Input	Output for the Sample Input

12
5
13
0
	4 9
	3 4
	1 9

D: Clock Splitter

Often a good approach is to start with a naive and correct solution, and then figure out how to speed it up.

First recall that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Now observe that for a given range which we cut off a b, that the difference between the two halves can be calculated in constant time. Let $s1$ denote side 1, which has sum

$$s1 = \sum_{i=a}^{b} i = \sum_{i=1}^{b} - \sum_{i=1}^{a-1} = \frac{b(b+1)}{2} - \frac{(a-1)a}{2}$$

and the other side $s2$ must have the sum of all numbers minus this

$$s2 = \sum_{i=1}^{n} i - s1 = \frac{n(n+1)}{2} - s1$$

and so we can calculate both $s1$ and $s2$ in constant time from a and b.

With $N = 1e5$, an $\mathcal{O}(N^2)$ algorithm which tries all possible pairs for the line to cut will be too slow.

But suppose for some b_i we have the optimal a_i, then for some $b_j > b_i$, we know that $a_j \geq a_i$. And so we can use the two pointer method. We will brute force the values for b, and find the next best value for a in amortised constant time.

The way the 2 pointer method works, is although there are $\mathcal{O}(N^2)$ possible ranges for a and b to represent, each pointer is only increased N times, so the total number of ranges that we will actually look at is $2N$, and we look at the ranges necessary to determine the answer.

You also need to be aware that $1e5 * (1e5 + 1)/2$ does not fit in a 32-bit integer, and so we will need to use 64-bit integers.

The time complexity of this solution is $\mathcal{O}(N)$ per test case.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

// suppose the line cuts from a to b on one side, what is the difference in sum between the two sides
ll dif(ll a, ll b, ll n){
  ll s1 = b*(b+1)/2 - (a-1)*a/2;
  ll s2 = n*(n+1)/2 - s1;
  return abs(s1 - s2);
}
```

```
int main(){
  ll n;
  while( (n=getint()) ){
    // initialise ret with a large enough value that we know we will find a better answer
    ll reta, retb, ret=n*(n+1)/2;
    ll a=1, b;
    // brute force b
    for(b=1; b<=n; ++b){
      // move a while ever it is profitable - N.B. < is necessary as we want the minimal a
      // that is, if two values of a produce the same difference, we want the smaller value of a
      while(dif(a+1, b, n) < dif(a, b, n))
        ++a;
      // try this combination of a and b
      // the < means we will only keep the first a,b pair with minimal cost as required
      // again, this is necessary to get the least possible a
      if(dif(a, b, n) < ret){
        ret = dif(a, b, n);
        reta = a;
        retb = b;
      }
    }
    // output
    printf("%lld %lld\n", reta, retb);
  }
  return 0;
}
```

Problem E

Neocortex Order

Time Limit: 20 seconds

The surface of the neocortex is partitioned into regions, not unlike the map of the Holy Roman Empire circa 1789. The neocortex in each region, which is approximately 3 mm thick, has six layers. The regions are connected by bundles of fibres that form the white matter of the brain. Each fibre starts in one region and sends signals in one direction to another region where it terminates. The starting and terminating layers may be different.

The connectivity from a region P to a region Q, which we shall describe by P→Q, is classified into one of the following three classes:
* <u>Ascending</u> 'A' if fibres from P terminate in layer four of Q.
* <u>Descending</u> 'D' if fibres from P terminate in layers of Q other than layer four.
* <u>Lateral</u> 'L' if fibres from P have terminations in all layers of Q.

For the visual and auditory cortices, it should be possible to arrange the regions into a hierarchy of discrete levels, such that for regions P and Q:

- P and Q are placed on the same level if either P→Q or Q→P are classified as 'L'.
- P is placed on a strictly lower level than Q if P→Q is classified as 'A' or if Q→P is classified as 'D'.
- P is placed on a strictly higher level than Q if P→Q is classified as 'D' or if Q→P is classified as 'A'.

Due to inconclusive experimental data for a given neocortex, the resulting classification of a number of connections may be in disagreement with any possible hierarchy. An 'L' connection is in disagreement with a hierarchy if the regions are not on the same level, an 'A' connection is in disagreement if the first region is not on a strictly lower level than the second region, and a 'D' connection is in disagreement if the first region is not on a strictly higher level than the second region.

Researchers have observed that no more than **five (5)** disagreements may occur. However they would like as few disagreements as possible to be removed to allow for the regions to be arranged into a hierarchy. Your task is to write a program to determine the minimum such number.

Input

The input consists of several test cases. Each test case starts with an integer N, 1 ≤ N ≤ 25, on a line by itself.

Each of the following N lines consists of three parts: a string STR1, followed by a letter T and then another string STR2. The three parts are separated by single spaces, and the two strings consist of letters and digits. The STR1 and STR2 correspond to two different regions, say r1 and r2. The letter T has one of the values 'A', 'D', 'L' to indicate the connection type from r1 to r2. Each line is unique within a test case.

A zero on a line by itself declares the end of input data and should not be processed.

Output

The output consists of a single line, for each test case, which contains the smallest number of connections in disagreement, over all possible arrangements of the regions into a hierarchy.

Sample Input	Output for the Sample Input
7 V2 A MT MT D V2 V2 A V4 V4 D V2 V4 A MT V4 L MT MT L V4 7 V2 A MT MT D V2 V2 A V4 V4 D V2 V4 A MT V4 D MT MT L V4 2 MT A MSTd 36 D 7a 0	1 2 0

E: Neocortex Order

First, what the question is really asking: You are given a directed graph, possibly with self loops and multiple edges, and also some edges actually mean that 2 vertices are on the same level. And you must find the minimum number of edges to remove such that the graph becomes acyclic.

Again, you generally want to start by asking yourself what is the naive solution. And in this case it is to try all possible

$$\sum_{i=0}^{4} \binom{25}{i} = 15276$$

ways of removing less than 5 edges. And since you are guaranteed that the answer is at most 5, if all of those fail you know that there must be a way with removing 5 edges and so you don't even have to find it.

So you literally construct all of those graphs and test them for being acyclic.

How to implement the solution. First you want to turn all of those string names for vertices into integers. In my solution this is done with the function *getid*.

Then you want to generate all subsets of edges of size $0 \ldots 4$. This is probably easiest done with a recursive function, though my preference is to use bitsets, as I know how to write a function to generate the lexicographically next bitset (in my solution I actually generate the bitsets of size $m \ldots m-4$, that is, the edges that are included, as opposed to excluded).

Now that you know what edges are going to be in the graph that you construct, it is time to construct the graph. First you establish the equivalency classes (vertices that are on the same level, we merge them into a single node in our graph). The constraints in this problem are so low that you could do this any way that you want (Floyd-Warshall or BFS/DFS), my preference is to use Union-Find as the code is short and easy to understand and it provides a really nice interface.

Now that lateral edges are taken care of, it is just a matter of constructing the graph on directed edges and testing for cycles. To construct the graph - go over all edges that are not being excluded, and are not lateral, and place an edge between the nodes in the new graph representing their equivalency classes (personally I decided to use coordinate compression to relabel the equivalency classes $0 \ldots x-1$ where x is the number of equivalency classes, though this is unnecessary, you could just have a bunch of isolated nodes in the graph and it would not be a problem).

Finally to check for cycles we use DFS. Perform the DFS and mark each node as being on the stack. If you see a vertex again while it is on the stack, then you have found a path from that vertex to itself. When you are finished DFSing from a vertex, mark it is seen and so never bother visiting it again.

This solution has time complexity $\mathcal{O}((\sum_{i=0}^{4} \binom{N}{i}) * N \lg N)$. Per graph that is constructed there are $O(N)$ vertices, and the sorting and calls to lower_bound dominate the time complexity. The dfs is linear and union find is $\mathcal{O}(N \lg^* N)$.

Some useful test cases for this problem:

```
Input
-----------
7
V2 A MT
MT D V2
```

```
V2 A V4
V4 D V2
V4 A MT
V4 L MT
MT L V4
7
V2 A MT
MT D V2
V2 A V4
V4 D V2
V4 A MT
V4 D MT
MT L V4
2
MT A MSTd
36 D 7a
1
X A X
1
X A Y
1
X L X
1
X L Y
4
X A Y
X A Y
X D Y
Y A X
7
P A Q
P A S
P A R
Q A S
Q A R
S A R
R A P
5
P A Q
Q A R
R A P
X A Y
Y A X
0

Expected output
--------------------
1
2
0
1
0
0
0
2
1
2

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

```
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }


// lexicographically next bitset with the same number of bits set
int nextComb(int x){ // x = xxx0 1111 0000
  if(!x){
    fprintf(stderr, "nextComb(0) called\num_vert");
    return ((1<<30)-1+(1<<30));
  }
  int smallest = x&-x;        // 0000 0001 0000
  int ripple = x+smallest;    // xxx1 0000 0000
  int ones = x^ripple;        // 0001 1111 0000
  ones = (ones>>2)/smallest;  // 0000 0000 0111
  return ripple|ones;         // xxx1 0000 0111
}

map<string,int> cache;
int getid(string s){
  if(!cache.count(s)){
    int t = sz(cache);
    cache[s] = t;
  }
  return cache[s];
}

// Union-Find - A disjoint set data structure
const int N = 1<<6;
int P[N], R[N];
void init(int n){
  for(int i=0; i<n; ++i){
    P[i] = i;
    R[i] = 1;
  }
}
int findP(int x){
  if(P[x] != x)
    P[x] = findP(P[x]);
  return P[x];
}
int merge(int a, int b){
  int pa=findP(a), pb=findP(b);
  if(pa == pb)
    return 0;
  if(R[pa] < R[pb])
    P[pa] = P[pb];
  else{
    P[pb] = P[pa];
    if(R[pa] == R[pb])
      ++R[pa];
  }
  return 1;
}

// O(n) cycle detection - This function returns 1 if a cycle is found, and 0 otherwise
// seen[i] = 0  -->  the node has not yet been visited
// seen[i] = 1  -->  the node is currently on the stack
// seen[i] = 2  -->  the node is not on the stack, and has been completely expanded
int tarjan(int i, vector< vector<int> > &a, vector<int> &seen){
  if(seen[i] == 1)
    return 1;
  if(seen[i] == 2)
    return 0;
  seen[i] = 1;
  for(int j=0; j<sz(a[i]); ++j)
    if(tarjan(a[i][j], a, seen))
      return 1;
  seen[i] = 2;
  return 0;
}

int main(){

  int num_edges;
  while( (num_edges=getint()) ){

    // read in the initial graph
    cache = map<string,int>();
    vector< pair< pair<int,int>,int > > b; // A->B, ?L
    for(int i=0; i<num_edges; ++i){
      string sx, sy, sz;
      cin >> sx >> sy >> sz;
      int x=getid(sx), z=getid(sz);
      if(sy == "D")
        swap(x, z);
      b.push_back(mp(mp(x,z),sy=="L"));
    }
    int num_vert = sz(cache);
    // start taking things out of it
    for(int drop=0; drop<5; ++drop) // brute force the number of things to drop (0..4)
      for(int in=(1<<(num_edges-drop))-1; in<(1<<num_edges); in=nextComb(in)){ // try all bit sets with drop bits missing
        // construct equivalency classes
        init(num_vert);
        for(int i=0; i<num_edges; ++i)
          if( in&(1<<i) && b[i].y )
            merge(b[i].x.x, b[i].x.y);
        // perform coordinate compression - this is not necessary
        vector<int> classes;
        for(int i=0; i<num_vert; ++i)
          if(findP(i) == i)
            classes.pb(i);
        sort(classes.begin(), classes.end());
        // construct the graph that is meant to be a dag
        vector< vector<int> > a(sz(classes));
        for(int i=0; i<num_edges; ++i)
          if( in&(1<<i) && !b[i].y ){
#define all(x) (x).begin(), (x).end()
            int from = lower_bound(all(classes), findP(b[i].x.x))-classes.begin();
            int to = lower_bound(all(classes), findP(b[i].x.y))-classes.begin();
            a[from].push_back(to);
          }
        // test if it is a dag
        // find all nodes with no predecessor
        vector<int> noIncoming(sz(a), 1);
        for(int i=0; i<sz(a); ++i)
```

```
          for(int j=0; j<sz(a[i]); ++j)
            noIncoming[a[i][j]] = 0;
        // dfs from them checking we don't find any cycles
        int ok = 1;
        vector<int> seen(sz(a), 0);
        for(int i=0; i<sz(a); ++i)
          if(noIncoming[i])
            if(tarjan(i, a, seen))
              ok = 0;
        // check we visited everybody, it may be the case that we don't even check people because
        // they are in a cycle with no on ramps
        for(int i=0; i<sz(a); ++i)
          if(!seen[i])
            ok = 0;
        // if we never find a cycle, we are done
        if(ok){
          printf("%d\n", drop);
          goto next_case;
        }
      }
    printf("5\n");

next_case:
    continue;


  }
  return 0;
}
```

Problem F
Some Pretty Peculiar Cells

Time Limit: 3 seconds

Samantha is a graduate student who has been studying a new variety of cells, which she named SPP (Some Pretty Peculiar) Cells. Samantha's first observations of SPP cells led her to believe that they are benign. That is, they neither increase nor decrease their number over time. Further investigations, by a co-researcher who wanted to duplicate her results, showed SPP cells to be a very aggressive type of cells that double their number every minute. Now Samantha has to explain her conclusions about SPP cells being benign to her supervisor who is very unhappy about the time he has wasted on a grant application based on her results.

Samantha has a theory to explain her data, which goes as follows:

Each SPP cell occupies exactly one (1) square micrometre of space in the rectangular Petri dish. As the population of SPP cells grows exponentially fast, they very quickly occupy all the space in the Petri dish (i.e., there are as many cells as the area of the dish). When this happens, a large number of the cells die from overcrowding.

The way the cells divide and die is very unique according to the theory. When the space in the dish does not allow for all the cells to divide, a number of cells go into hibernation for one (1) minute. Each of the remaining cells duplicates, which makes the total number of cells equal to the Petri dish's area. The newly born cells then die immediately from overcrowding. In the next minute, all the hibernating cells wake up and continue their exponential growth.

For example, consider a rectangular Petri dish with an area of 100,000 square micrometres that contains 30,000 SPP cells at the starting time T = 0. At 1 minute, every cell splits and there will be 60,000 SPP cells. In the second minute 40,000 SPP cells will duplicate and 20,000 will hibernate, which brings the total number of cells to 100,000. The 80,000 cells that were just born will die, from overcrowding, leaving just 20,000 SPP cells at the beginning of the second minute. At the third minute the disc will have 40,000 SPP cells, and so on.

Your task is to write a program to verify Samantha's theory. Your program should report the smallest interval of time T, where T > 0, at which time the Petri dish contains the same number of SPP cells as it has at time T = 0. If there is no such interval, the program should report IMPOSSIBLE!

Input

The input starts with an integer N, on a line by itself, that represents the number of cases. $1 \leq N \leq 1000$. The description for each case consists of three integers, X, Y and A, separated by single blank spaces, on a line by themselves. X and Y represent the length of the rectangular Petri dish sides in micrometres, and A is the starting number of SPP cells in the Petri dish. $1 \leq X, Y, A \leq 40000$.

Output

The output consists of a single line, for each test case, which contains the shortest interval of non-zero time until the number of SPP cells is the same as it was at time 0. If there is no such interval print the string "IMPOSSIBLE!"

Sample Input	Output for the Sample Input

3
3 4 8
3 7 2
10000 10000 1
	2
	6
	IMPOSSIBLE!

F: Some Pretty Peculiar Cells

Let $m = XY$.

So the problem is, find the least t such that $A * 2^t \equiv A \pmod{m}$.

We first divide out by $gcd(A, m)$ (first observing that this does not affect the answer). To arrive at the problem, find the least t such that $A' * 2^t \equiv A' \pmod{m'}$.

Now that they are coprime, A' has an inverse and we can cancel the A''s. So we desire the least t such that $2^t \equiv 1 \pmod{m'}$.

We then use Euler's theorem

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

where

$$\phi(n) = n \prod_{p|n} \frac{p-1}{p}$$

So we have

$$2^{\phi(m')} \equiv 1 \pmod{m'}$$

Theorem:
The least t such that $2^t \equiv 1 \pmod{m'}$ divides $\phi(m')$.

Proof:
Let x be the minimal element of $\{\ t \mid 2^t \equiv 1 \pmod{m'}\ \}$. And suppose x does not divide $\phi(m')$, then $\phi(m') = kx + r, 0 < r < x$. Now

$$1 \equiv 2^t \equiv 2^{kx+r} \equiv 2^{kx} * 2^r \equiv (2^x)^k * 2^r \equiv 1^k * 2^r \equiv 2^r \pmod{m'}$$

That is, $2^r \equiv 1 \pmod{m'}$. But $r < x$, contradicting that x was the minimal element of the set.

Therefor our supposition that x does not divide $\phi(m')$ must have been wrong, and so x does divide $\phi(m')$. ■

So the solution is to brute force over all divisors of $\phi(m')$ to find the minimal one with the required property.

Time complexity:

In the worst case factorise walks all the way up to the square root of the input. Apart from the call to factorise, the totient function runs in logarithmic time. Gcd runs in logarithmic time. ApowBmodC runs in time logarithmic in b. And in the main function we only walk up to the square root of the input. The call to ApowBmodC only happens for factors, so less than $2 * \sqrt{n}$ times, and this gives us a bound of $\mathcal{O}(\sqrt{m} \lg m)$ per test case.

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

```
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

vector< pair<int,int> > factorise(int x){
  vector< pair<int,int> > ret;
  for(int i=2; i*i<=x; ++i) if(x%i == 0){
    int cnt = 0;
    while(x%i == 0){
      ++cnt;
      x /= i;
    }
    ret.push_back(mp(i, cnt));
  }
  if(x != 1)
    ret.push_back(mp(x, 1));
  return ret;
}

int totient(int x){
  vector< pair<int,int> > a = factorise(x);
  int ret = x;
  for(int i=0; i<sz(a); ++i){
    ret /= a[i].x;
    ret *= a[i].x-1;
  }
  return ret;
}

int gcd(int a, int b){ return b ? gcd(b, a%b) : a; }

int ApowBmodC(ll a, int b, int c){
  if(b==0) return 1;
  int ret = ApowBmodC(a*a%c, b/2, c);
  if(b&1) ret = ret * a % c;
  return ret;
}

int main(){
  int ttt=getint();
  for(int tt=0; tt<ttt; ++tt){
    int m=getint()*getint(), a=getint();
    // make a and m coprime
    int g = gcd(a, m);
    a /= g;
    m /= g;
    // find the totient of m
    int i, t=totient(m);
    // brute force all divisors of the totient
    for(i=1; i*i<=t; ++i)
      if(t%i == 0 && ApowBmodC(2, i, m) == 1){
        printf("%d\n", i);
        goto next;
      }
    for(i=i; i>0; --i)
      if(t%i == 0 && ApowBmodC(2, t/i, m) == 1){
        printf("%d\n", t/i);
        goto next;
      }
    // when m is even, or a==m, the above will fail
    printf("IMPOSSIBLE!\n");
next:
    continue;
  }
  return 0;
}
```

Problem G
Mealy's Memoirs

Time limit allowed: 2 seconds

After a distinguished career as a computer scientist, Reginald Mealy wrote his memoirs and used a speech recognition program to transcribe his life story as he spoke. The result was many large streams of text.

However, all did not go according to plan because the memoirs contained the email addresses of many of his associates. The speech recognition programs transcribed the email addresses literally such that me@me.com was transcribed as meatmedotcom. All special characters were transcribed literally: "@" was transcribed as "at" and "." was transcribed as "dot". As well, spaces were omitted unless there was an unusually long pause.

Reginald Mealy wants to extract the email addresses from the transcribed text, and he has set it as a task for his student. As an incentive, he has offered a bounty of 128 cents for each extracted email address.

The student, being greedy, would like to maximise the number of extracted email addresses, and would like you to write a program to calculate the maximum bounty that can be collected for a stream of text. You may assume that all email addresses have one of the following forms:

[word]@[word].[word]
[word]@[word].[word].[word]
[word]@[word].[word].[co]
[word]@[word].[word].[word].[co]

where

[word] is a word of between 3 and 16 lower case Roman letters, and
[co] is a two letter country code,
where all two-letter combinations are valid country codes.

So the following two strings aaa@aaa.aaa and aaa@aaa.aaa.aaa.aa are valid email addresses, but the following three strings a@a.a, @a or a@a. are not valid. Note that a bounty is collected only once if multiple email addresses overlap.

Input

The input starts with an integer N, on a line by itself, which indicates the number of test cases. $1 \leq N \leq 100$. The description of each test case consists of a string made entirely of lower case Roman letters from a to z, inclusive, and the space character.

Output

For each test case the output consists of a single line that contains the maximum bounty that can be received. The amount is to be given in the format of $d.cc, where d is the number of dollars and cc is the number of cents (with a leading 0 if required).

Sample Input

```
6
aaaaaaaameatmeedotcombbbbbbbb
bbbbmattatdiedotdotnetpatdotdotherityatiinetdotnetaaaa
myemailathomedot com
myemailatabcdefghijklmnopqrstuvwxyzdotcom
helmattatdidotdotnet
helmatdotdotherityatiinetdotnet
```

Output for the Sample Input

```
$1.28
$3.84
$0.00
$0.00
$1.28
$2.56
```

G: Mealy's Memoirs

Theorem:
We only need to look for emails matching the pattern $[a-z]\{3\}at[a-z]\{3-16\}dot[a-z]\{3\}$.

Proof:
First observe that this pattern is a substring of all acceptable email forms, and is itself an acceptable email. So trivially a maxmimal disjoint set of substrings matching this pattern is a lower bound for the solution to the problem, as it is itself a solution.

Now consider a solution to this problem with a maximal number of disjoint emails. You can then take the substring of each of these emails which only matches the above pattern, and you will have another solution to the problem with the same number of disjoint emails all matching the pattern. And the bound goes the other way as well, and we have equality. ■

From here, the algorithm is greedy (I thank Ben Dean for pointing this out to me on the SPPContest facebook group page).

First, break the lines up by whitespace (emails cannot contain whitespace, and so must be contained completely within a block of consecutive lower case roman letters). Then search for "*at*"'s, if you find one you then search for a "*dot*" within an appropriate distance, and count the email.

Theorem:
The greedy algorithm is correct.

Proof:
It can be argued inductively that our algorithm has found the maximum amount of emails possible in all prefix's of the input. Base case is the empty prefix, which our algorithm says there are no emails yet and this is the maximum number. For the inductive case, suppose it is possible at some point j to have found more emails than what our algorithm has found, and that the last email in this better solution started at i. Then consider the prefix of the input up to i, by induction our algorithm is not already beat at this point, but given that our algorithm is equal with this better solution at point i, it contradicts the mechanics of our algorithm that we are then beat at j, as we will find this one extra email. ■

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

int main(){
  int ttt=getint();
  string s;
  getline(cin, s); // discard the remainder of the line with the number of test cases on it
  for(int tt=0; tt<ttt; ++tt){
    getline(cin, s);
    stringstream ss;
    ss << s;
    int ret = 0;
    // break up on whitespace, as emails may not contain whitespace
    while(ss >> s){
      int istart = 3; // 3 for s_1;
      int iend = sz(s)-10; // 3 for s_3, 3 for dot, 3 for s_2, 1 for the t in at
      for(int i=istart; i<iend; ++i){ // greedily look for at
        if(s[i]=='a' && s[i+1]=='t'){
          int jstart = i+5; // 2 for at, 3 for s_2
          int jend = min(i+18+1, sz(s)-5); // 2 for at, 16 for s_2, 1 for exclusive  --  3 for s_3, 2 for "ot"
          for(int j=jstart; j<jend; ++j)
            if(s[j]=='d' && s[j+1]=='o' && s[j+2]=='t'){
              ++ret; // we found an email
              // skip over all letters used in this email before trying to start a new email
              i = j+9-1; // 3 for dot, 3 for s_3, 3 for new s_1, subtract 1 because the i-loop will increase i
              break;
            }
        }
      }
    }
    // the maximum number of emails
    ret *= 128;
    printf("$%d.%02d\n", ret/100, ret%100);

  }
  return 0;
}
```

Problem H
Ninja Pizza

Time Limit: 3 seconds

Ninja Pizza is getting popular because its delivery boys wear ninja costumes pretending to be real ninjas who are renowned for their excellent sword skills. The company advertisements say that a ninja can easily slice two of their perfectly round pizzas, when placed with their interiors non-overlapping on a flat table, into two equal-area halves with one swing of their sword.

The delivery boys are not real ninjas and the shape of each pizza is not really round, but is in the form of a convex polygon with N vertices. $3 \leq N$.

Your task is to write a program that calculates the necessary sword stroke, which a real ninja would instinctively do, for a given two pizzas. A sword stroke is described by a straight line in the form of $y = m * x \pm b$ in the x-y plane.

Input

The input starts with an integer C, on a line by itself, that represents the number of test cases. $1 \leq C$. Each test case contains the specifications of two convex polygons. The first line of a polygon specification consists of an integer N that represents the number of its vertices. Each of the following N lines contains two real numbers, separated by a single space, that are the x- and y-coordinates of one vertex, respectively. The vertices are given in the anti-clockwise order.

The total number of vertices over all test cases is not more than 5000 and values of all x- and y- coordinates are in the range from -1 to 1, inclusive.

There will always be a solution for each case with $-1000 \leq m \leq 1000$.

Output

For each test case the output consists of a single line that contains the equation of a planar straight line that bisects both polygons. The equation is to be printed in the familiar form of "y␣=␣m␣x␣±␣b", where the symbol "␣" indicates a single space and "±" indicates the sign of b.

The values of m and b must be printed as shown in the sample data below, after rounding off to the hundredths place.

Sample Input	Output for the Sample Input

1
4
0.0 0.2
0.1 0.2
0.1 0.3
0.0 0.3
4
0.1 0.3
0.2 0.3
0.2 0.4
0.1 0.4
	y = 1.00 x + 0.20

Printing of zero

The value of zero should always be printed as 0.00
An output of "-0.00" will result in a WRONG-ANSWER and 20 minutes penalty.

Rounding off is used to approximate the value of a number to a specified decimal place. After rounding, the digit in the place we are rounding will either stay the same, referred to as rounding down, or increase by 1, referred to as rounding up. Examples are:

if we round 6.734 to the hundredths place, it is rounded down to 6.73.
if we round 6.735 to the hundredths place, it is rounded up to 6.74.
if we round -6.734 to the hundredths place, it is rounded up to -6.73.
if we round -6.735 to the hundredths place, it is rounded down to -6.74.

H: Ninja Pizza

The solution is nested binary search. For a fixed m, we can binary search for the b such that $y = mx + b$ cuts the first polygon in half. And the outer binary search for m is searching for the gradient such that when you cut the first polygon in half, you also cut the second polygon in half.

The time complexity of this solution is $\mathcal{O}(P * T^2)$, where P is the number of points, and in this case T is the number of times that I have decided to run the loop of the binary search, which is 100. I chose 100 because doubles have 64-bits and so I will get very close to the most accurate double. I do not run the loop until the bounds become within some tolerance because in certain circumstances doubles are not able to converge within the tolerance and the program infinite loops.

The following tutorial should help in understanding the wonders of the cross product:

`http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry1`

In particular you will want to understand the presented solution to finding the area of a polygon.

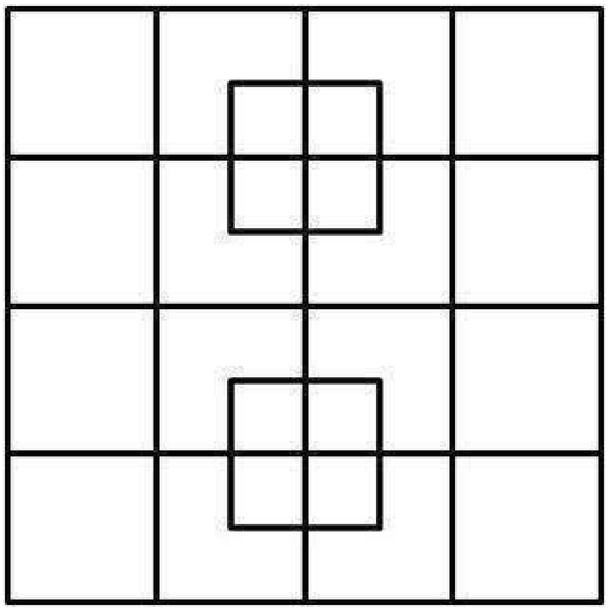
```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }
inline double getdouble() { double a; scanf("%lf", &a); return a; }

const double EPS = 1e-9;

typedef pair<double,double> pt;

// given polygon p, and line y = mx + b, find area above - area below
double area(vector<pt> &p, double m, double b){
  double ret = 0;
  for(int i=0; i<sz(p)-1; ++i){
    double tc = p[i].y - m*p[i].x - b;
    double td = p[i+1].y - m*p[i+1].x - b;
    double r;
    if(min(tc,td) > 0) // triangle completely above the line
      r = 1;
    else if(max(tc,td) < 0) // triangle completely below the line
      r = -1;
    else
      r = 2*max(tc,td)/(max(tc,td)-min(tc,td)) - 1; // other wise weight by amount above/below
    ret += r * (p[i].x*(p[i+1].y-b) - p[i+1].x*(p[i].y-b)); // this is cross product, area of triangle
  }
  return ret;
}

// here is the inner binary search, we search for b that cuts the polygon in half
// that is the area above - area below = 0
double bisect(vector<pt> &p, double m){
  double lo = -1e9, hi = 1e9;
  for(int k=0; k<100; ++k){
    double mid = (lo+hi)/2;
    if(area(p, m, mid) > 0)
```

```
      lo = mid;
    else
      hi = mid;
  }
  return lo;
}

int main(){
  int ttt=getint();
  for(int tt=0; tt<ttt; ++tt){

    // input
    vector<pt> a[2];
    for(int i=0; i<2; ++i){
      int n=getint();
      for(int j=0; j<n; ++j){
        double x=getdouble(), y=getdouble();
        a[i].pb(pt(x,y));
      }
      a[i].pb(a[i][0]); // add the first point again, now all consecutive pairs of points is all segments
    }

    // for want of a better description, the area being pos/neg means the gradient needs to go up or
    // down depending upon which polygon is on the left and which on the right. Where on the left
    // and on the right is only one of the cases that this really handles.
    double hi = 1e3, lo = -1e3;
    if(area(a[1], 1e3, bisect(a[0], 1e3)) <= 0)
      swap(hi, lo);

    // perform nested binary search, here we search for m
    for(int k=0; k<100; ++k){
      double mid = (hi + lo) / 2;
      if(area(a[1], mid, bisect(a[0], mid)) > 0)
        hi = mid;
      else
        lo = mid;
    }

    // we have the answer now
    double retm = lo;
    double retb = bisect(a[0], retm);

    // adjust m and b for printing to make sure the rounding happens properly
    if(retm > -EPS)
      retm = abs(retm);
    if(retb > -EPS)
      retb = abs(retb);
    if(retm >= 0)
      retm += EPS;
    else
      retm -= EPS;
    if(retb >= 0)
      retb += EPS;
    else
      retb -= EPS;

    // output
    printf("y = %.02lf x %c %.02lf\n", retm, retb>=0 ? '+' : '-', abs(retb));

  }
  return 0;
}
```

Problem I
Count the Squares

Time Limit: 15 seconds

The puzzle “*How many squares can you count in this image*?”



has recently been popular amongst members of some social networking sites. A posting of such a puzzle is usually followed by an endless number of likes, replies with possible answers, and arguments.

Posting a code that calculates the correct number of squares for such a puzzle would be a great spoiler. Your task is to write a program to solve this type of puzzle.

By the way, the answer for the above puzzle, which is an 8x8 grid, is **40**.

Input

The input starts with an integer N, on a line by itself, that represents the number of puzzles. 1 ≤ N ≤ 100.

The description for each puzzle starts with an integer L, on a line by itself , that represents the total number of line segments that form the puzzle. The set of line segments are rectilinear. That is, its elements are parallel to the x- or y-axis. 0 ≤ L ≤ 2000.

Each of the following L lines contain 4 integers, separated by single blank spaces, that describe a single line segment. The first two integers are the x- and y-coordinates of one end, and the second pair of numbers are the x- and y-coordinates of the other end. Values of the x- and y-coordinates are in the range of 1 to 10000, inclusive.

Output

The output consists of a single line, for each test case, which contains an integer that represents the total number of squares.

Sample Input	Output for the Sample Input

2
9
1 1 1 2
1 2 1 3
1 2 2 2
2 1 2 2
2 2 2 3
2 1 3 1
2 2 3 2
3 1 3 2
3 2 3 3
8
1 1 4 1
1 1 1 4
1 4 4 4
4 1 4 4
3 3 6 3
3 3 3 6
3 6 6 6
6 3 6 6
	1
	3

I: Count the Squares

Let $L = 2000$ be the number of segments, $M = 10,000$ be the range of coordinates, and $N = 100$ be the number of test cases.

First, merge overlapping/touching segments. This then allows us to record for each point, how far up/right/down/left that we can go from that point, staying on segments.

Observe that (x, y) can only be a corner of a square if there is a vertical segment with x-coordinate x, and a horizontal segment with y-coordinate y. So we can only get one coordinate that we care about for trying the corners of squares for each segment. So if X is the number of x-coordinates we care about, and Y the number of y-coordinates, then $X + Y \leq L$, and so we care about at most $\frac{L}{2} * \frac{L}{2} = \frac{L^2}{4}$ points for being corners.

Let's consider the task of counting all the squares one by one. We could now just say, for all possible bottom left corners, for all possible top right corners (so only check up and right of the previous point), test if this is a square by checking that:

- The x and y distance between the two points is equal, let this distance be d.
- Both the distance that you can go up, and can go right from the bottom left point must be at least d.
- Both the distance that you can go down, and can go left from the top right point must be at least d.

But there could be $\mathcal{O}(L^2)$ intersections, and so naively we would test $\mathcal{O}(L^4)$ pairs of points. And this is too slow.

Observing that the given the first point and the x-coordinate of the second point, we uniquely determine the y-coordinate of the second point. And so this solution would be $\mathcal{O}(L^3)$. Still too slow.

Lets see what a maximal answer could be. Suppose there were 1000 segments with coordinates $(x, 1, x, 1000)$ for x in the range $1 \ldots 1000$, and 1000 segments with coordinates $(1, y, 1000, y)$ for y in the range $1 \ldots 1000$. Then we count for each size of square i, that there are $(1001 - i)^2$ squares with that size, which gives us

$$\begin{aligned} \sum_{i=1}^{1000} (1001 - i)^2 &= \sum_{i=1}^{1000} i^2 \\ &= \frac{2 * i^3 + 3 * i^2 + i}{6}, \text{ for } i = 1000 \\ &= 333,833,500 \end{aligned}$$

With 100 test cases, it is not going to be possible to count $3e8$ squares 100 times, counting them one by one. So in fact, any solution which only counts the squares one at a time will be too slow. So we try to think of an approach which will allow us to count many squares at time.

The approach that this solution will take: for all possible intersections (of which there could be $\mathcal{O}(L^2)$), suppose that it is the bottom left corner of a square, count the number of intersections that form a top right corner (of which there could be $\mathcal{O}(L)$, in time $\mathcal{O}(\lg L)$ using a binary indexed tree (BIT)).

In a nutshell, a BIT is an array to which you can add and remove things in logarithmic time, and query the sum of everything in the BIT up to an arbitrary point.

For those of you not familiar with BIT's I refer you to:

`http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees`

`http://e-maxx.ru/algo/fenwick_tree`

in this case I by far recommend the top coder tutorial, by excluding the cell 0 from the BIT the arithmetic is much easier.

For a given bottom left corner, all possible top right corners have the same $x - y$, so we form all $\mathcal{O}(L^2)$ intersections and process them by diagonal. On each diagonal we process them from top right to bottom left. Now that we are on a single diagonal, this is equivalent to the one dimensional case where each point has a distance that it can reach to the right (in the original 2D problem, the minimum of how far it can reach up and to the right) and how far it can reach to the left (in the original 2D problem, the minimum of how far to the left and down it can reach).

Each point queries how many squares it is the bottom left of by querying the BIT with the value of it's x-coordinate + how far to the top right it can reach. And in the BIT we store for each point it's x-coordinate (thus somebody querying the BIT will only count this point if they can reach it), and we remove a point from the BIT once it can no longer reach the point querying the BIT (we do this using a priority queue ordered by the time we delete the point, and associating with it the point in the BIT to be deleted).

The time complexity of this solution is $\mathcal{O}(NL^2 \lg L)$.

Honestly I do not recommend reading the code that follows, I include it only for completeness. All the coordinate compression stuff makes it hideously ugly. I am assuming that the intended solution was much more elegant than my own.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

/* Coordinate compression, now there are at most L^2 intersections.
 * Sort in a bunch of difference directions to find furthest each point reaches.
 * Then sort in x-y and x+y to do the querying.
 */

typedef pair<int,int> ii;

const int N = 1<<12; // max num unique coordinates
const int M = 1<<14; // max value of a coordinate
// map coordinates to their compressed coordinates
int ix[M], iy[M];
// these are indexed by the compressed coordinates of a point, and return the compressed coordinate
// highest x reachable along segments from this point, highest y ... etc.
int hx[N][N], hy[N][N], lx[N][N], ly[N][N];

// queries the sum of the range 1..i in logarithmic time
int bit[M+1];
int query(int i){
  int ret = 0;
  while(i){
    ret += bit[i];
    i -= i&-i;
  }
  return ret;
}
void update(int i, int x){
  while(i <= M){
    bit[i] += x;
    i += i&-i;
  }
}

// least .x.y, then least .x.x
int ycomp(const pair<ii,ii> &a, const pair<ii,ii> &b){
  return a.x.y != b.x.y ? a.x.y < b.x.y : a.x.x < b.x.x;
}

// least x-y, then greatest x
int xycomp(const ii &a, const ii &b){
  return a.x-a.y != b.x-b.y ? a.x-a.y < b.x-b.y : a.x > b.x;
}

int main(){
  int ttt=getint();
  for(int tt=0; tt<ttt; ++tt){

    // input
    memset(hx, -1, sizeof(hx));
    memset(hy, -1, sizeof(hy));
    memset(lx, -1, sizeof(lx));
    memset(ly, -1, sizeof(ly));
    int n=getint();
    vector< pair<ii,ii> > t1, t2, a;
    vector<int> xs, ys;
    for(int i=0; i<n; ++i){
      int x1=getint(), y1=getint(), x2=getint(), y2=getint();
      // make all edges point up and right
      if(x1 > x2 || y1 > y2){
        swap(x1, x2);
        swap(y1, y2);
      }
      t1.push_back(mp(ii(x1,y1),ii(x2,y2)));
    }

    // unify vertical segments
    sort(t1.begin(), t1.end());
    for(int i=0; i<sz(t1); ++i)
      if(t1[i].x.x == t1[i].y.x){
        int reach = t1[i].y.y, j;
        for(j=i+1; j<sz(t1) && t1[j].x.x==t1[i].x.x && t1[j].x.y<=reach; ++j)
          reach = max(reach, t1[j].y.y);
        a.push_back(mp(t1[i].x,ii(t1[i].x.x,reach)));
        i = j-1;
      }

    // unify horizontal segments
    sort(t1.begin(), t1.end(), ycomp);
```

```
for(int i=0; i<sz(t1); ++i)
  if(t1[i].x.y == t1[i].y.y){
    int reach = t1[i].y.x, j;
    for(j=i+1; j<sz(t1) && t1[j].x.y==t1[i].x.y && t1[j].x.x<=reach; ++j)
      reach = max(reach, t1[j].y.x);
    a.push_back(mp(t1[i].x,ii(reach,t1[i].x.y)));
    i = j-1;
  }

// coordinate compression
for(int i=0; i<sz(a); ++i){
  xs.push_back(a[i].x.x);
  xs.push_back(a[i].y.x);
  ys.push_back(a[i].x.y);
  ys.push_back(a[i].y.y);
}
sort(xs.begin(), xs.end());
xs.resize(unique(xs.begin(), xs.end())-xs.begin());
for(int i=0; i<sz(xs); ++i)
  ix[xs[i]] = i;
sort(ys.begin(), ys.end());
ys.resize(unique(ys.begin(), ys.end())-ys.begin());
for(int i=0; i<sz(ys); ++i)
  iy[ys[i]] = i;

// find how far everything can reach up, right, down, left
for(int i=0; i<sz(xs); ++i)
  for(int j=0; j<sz(ys); ++j){
    hx[i][j] = lx[i][j] = i;
    hy[i][j] = ly[i][j] = j;
  }
for(int i=0; i<sz(a); ++i){
  int _ix = ix[a[i].x.x];
  int _iy = iy[a[i].x.y];
  int _ix2 = ix[a[i].y.x];
  int _iy2 = iy[a[i].y.y];
  if(_ix != _ix2)
    for(int j=_ix; j<=_ix2; ++j){
      hx[j][_iy] = _ix2;
      lx[j][_iy] = _ix;
    }
  if(_iy != _iy2)
    for(int j=_iy; j<=_iy2; ++j){
      hy[_ix][j] = _iy2;
      ly[_ix][j] = _iy;
    }
}

// we no longer care about the segments, but we need all the points
vector<ii> pts;
for(int i=0; i<sz(xs); ++i)
  for(int j=0; j<sz(ys); ++j)
    pts.push_back(ii(xs[i], ys[j]));

// we will traverse all diagonals, so lets sort the points so those on the same x-y=c diagonal
// are together
sort(pts.begin(), pts.end(), xycomp);

ll ret = 0;
```

```
    // for all diagonals count the number of squares
    priority_queue< ii, vector<ii> > dieTime; // < time, x >
    for(int i=0; i<sz(pts); ++i){
      // clear the BIT if this is a new diagonal
      if(!i || pts[i].x-pts[i].y != pts[i-1].x-pts[i-1].y){
        while(!dieTime.empty()){
          update(dieTime.top().y, -1);
          dieTime.pop();
        }
      }
      // remove points that I am no longer able to query
      while(!dieTime.empty() && dieTime.top().x > pts[i].x){
        update(dieTime.top().y, -1);
        dieTime.pop();
      }
      // query how many points to the top right reach me, and I reach them
      int dx = xs[ hx[ ix[pts[i].x] ][ iy[pts[i].y] ] ] - pts[i].x;
      int dy = ys[ hy[ ix[pts[i].x] ][ iy[pts[i].y] ] ] - pts[i].y;
      ret += query(pts[i].x + min(dx,dy));
      // place me so people can query me
      update(pts[i].x, 1);
      dx = pts[i].x - xs[ lx[ ix[pts[i].x] ][ iy[pts[i].y] ] ];
      dy = pts[i].y - ys[ ly[ ix[pts[i].x] ][ iy[pts[i].y] ] ];
      dieTime.push(ii(pts[i].x-min(dx,dy), pts[i].x));
    }
    // clear the BIT
    while(!dieTime.empty()){
      update(dieTime.top().y, -1);
      dieTime.pop();
    }

    // output
    printf("%lld\n", ret);

  }
  return 0;
}
```

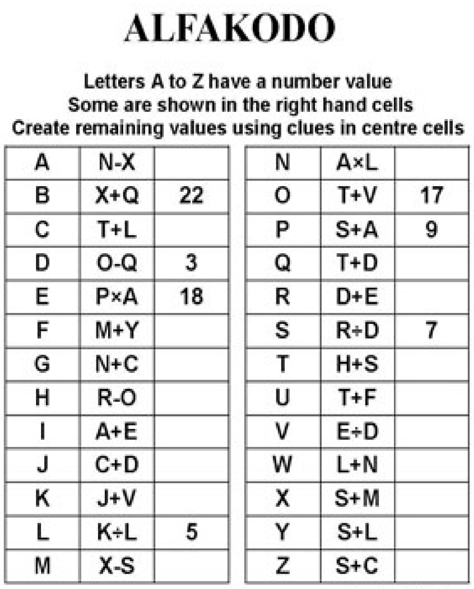
Problem J
yet Another Newspaper Puzzle

Time limit allowed: 5 seconds

Some newspapers attempt to increase their sales by offering puzzles that challenge the readers and occupy the time of their daily commute.

Alfakodo is one such puzzle that asks for unique numeric values to be assigned to the twenty-six letters of the English alphabet, subject to given constraints. Each constraint is given as a clue in the form of a simple arithmetic expression. For example, A = N-X.

Since the puzzle is not meant to frustrate the reader, the puzzle is constructed in such a way that throughout the solution process there is always at least one letter whose value can be ascertained.



ALFAKODO

Letters A to Z have a number value
Some are shown in the right hand cells
Create remaining values using clues in centre cells

A	N-X		N	A×L	
B	X+Q	22	O	T+V	17
C	T+L		P	S+A	9
D	O-Q	3	Q	T+D	
E	P×A	18	R	D+E	
F	M+Y		S	R÷D	7
G	N+C		T	H+S	
H	R-O		U	T+F	
I	A+E		V	E÷D	
J	C+D		W	L+N	
K	J+V		X	S+M	
L	K÷L	5	Y	S+L	
M	X-S		Z	S+C	

Input

The input starts with an integer N, on a line by itself, which indicates the number of test cases. $1 \le N \le 100$. The description of each test case consists of twenty six (26) lines, one line for each letter of the English alphabet. Each line consists of two or three parts, separated by single spaces:

1. The first part is a letter of the alphabet from A to Z, inclusive.
2. The second part is an arithmetic expression that consists of two letters separated by one of the four operations (+, - , / and *).
3. The third part, if present, is an integer value in the range of one (1) to fifty (50), inclusive.

Output

For each test case the output consists of a single line that contains the values of the alphabet letters, from A to Z, separated by single spaces.

Sample Input

```
1
A N-K
B Z-I
C A+N 21
D Y-W
E R+P 8
F I+U
G Y+I 39
H Q+R
I I/R 2
J Q-W 22
K I+P
L L*R
M H-F
N E+P
O C-V 5
P P/R
Q L-I
R H-Q
S F+W
T Y-B
U G-J
V V*R
W X-C
X N+V
Y G-I
Z C+P
```

Output for the Sample Input

```
6 26 21 27 8 19 39 33 2 22 9 34 14 15 5 7 32 1 29 11 17 16 10 31 37 28
```

J: Yet Another Newspaper Puzzle

I have not solved this problem. I have merely come to the conclusion that this problem is hard.

Consider a test case in which all variables are solved except *A*-*H*, and these are not mentioned in any of the other formulas. And the formulae for *A*-*H* are:

- A = D + E
- B = D - E
- C = A + B
- D = F + F
- E = C - G
- F = H + H
- G = F + D
- H = H * H

From an in contest clarifation, we know that all answers are positive integers. And so $H = 1$ (it could also be 0 without this clarification). And then, we know that $F = 2$, $D = 4$ and $G = 6$. And then we have to make the leap that $C = A + B = (D + E) + (D - E) = 2D = 8$ (though non-trivial, this value can be ascertained). And then the rest falls out. $E = 2$, $A = 6$, $B = 2$.

Also observe that this is just a simple demonstration of concept, much more complex test cases could be constructed in the same vein.

Problem K
Encyclopaedia of Equality

Time Limit: 15 seconds

Edith has planned for her “*Encyclopaedia of Equality*” to have the property that all volumes of the encyclopaedia have exactly the same length. As a result, editing one article occasionally raises the need for shuffling of articles between the volumes to restore the equality of lengths. Edith has found that the shuffling of articles makes it impossible to maintain the article names in alphabetically sequential order through the encyclopaedia. For example, a volume may contain the articles from *Cockroach* to *Demagogue* and from *Ethiopia* to *Finger* and an article on *Oysters*, while another volume may contain an article on *Democracy*.

Edith wants to name each volume with the minimum number of alphabetically sorted sequential article ranges, separated by “,␣”. A range is either a single article name, or the first and last article name in the range separated by “␣-␣”. The symbol “␣” indicates a single space. Examples of volume names are:

- *Democracy*
- *Cockroach␣-␣Demagogue,␣Ethiopia␣-␣Finger,␣Oyster*

Edith tried to manually update the volume names when she shuffled some articles, but this proved to be too laborious. Your task is to write a program that updates volume names after a reshuffle of articles.

Input

The first line of input contains two integers M and N, separated by a single space, on a line by themselves. M is the number of articles in the encyclopaedia and N is the number of test cases, where $0 < M < 1000000$ and $0 < N < 1000$.

The second line contains M words separated by single spaces and sorted in increasing alphabetical order. Each word is the name of an article in Edith's encyclopaedia and consists of a sequence of alphabetic characters.

The rest of the input contains a series of N test cases. The first line in each test case begins with the word START followed by an initial volume name. Subsequent lines consist of one of the following options:

- A line starting with ADD followed by the article ranges, separated by “,␣”, to be added to the volume. Articles to be added may already exist in the volume name.
- A line starting with REMOVE followed by the article ranges, separated by “,␣”, to be removed from the volume. Articles are not necessarily included in the volume before they are removed.
- A line that contains the single word END that indicates the end of the test case.

In each ADD or REMOVE line, there are at most 50,000 ranges. The ranges are not necessarily sorted alphabetically but the second article name in a range will always occur after the first article name in alphabetical order. Also the ranges are not necessarily specified minimally but they do not overlap.

Output

The output consists of a single line, for each test case, which contains the properly formatted minimum volume name after all the ADD and REMOVE updates have been performed.

Sample Input
10 2
Cockroach Costume Demagogue Democracy Demonstration Eagle Ethiopia Finger Helicopter Oyster
START Cockroach - Demagogue, Ethiopia - Finger, Oyster
ADD Democracy - Demonstration
REMOVE Oyster
END
START Costume - Eagle
REMOVE Eagle, Costume
END

Output for the Sample Input
Cockroach - Demonstration, Ethiopia - Finger
Demagogue - Demonstration

K: Encyclopaedia of Equality

There is not much to say for this problem, you just do what they told you to do, and it is a matter of knowing how to do it. You can use a balanced binary search tree to perform all operations in log time (c++'s set, java's treeSet).

Add the ranges one by one, and remove them one by one, and maintain the state of all segments presently in the data structure.

Let X be the total number of remove and add operations.

The total number of operations is bounded linear on the input because each range in the set can be added at most once, removed at most once, and edits to ranges can happen at most twice per query. So despite the fact that there are $\mathcal{O}(X)$ remove operations, and on a single remove operation we may remove $\mathcal{O}(X)$ segments, the overall bound is still $\mathcal{O}(X)$.

The time complexity of this solution is $\mathcal{O}(X \lg M)$, where M is the number of articles.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

map<string,int> cache;
vector<string> words;
int getid(string s){
  if(!cache.count(s)){
    cache[s] = sz(words);
    cache[s+","] = sz(words);
    words.pb(s);
  }
  return cache[s];
}

// read a line, and turn it into ranges
// all of these ranges will have x inclusive, and y exclusive
vector< pair<int,int> > parse(){
  string s;
  getline(cin, s);
  vector< pair<int,int> > ret;
  stringstream ss;
  vector<string> tokens;
  ss << s;
  while(ss >> s)
    tokens.push_back(s);
  int i = 0;
  for(int j=0; j<sz(tokens); ++j)
    if(tokens[j][sz(tokens[j])-1]==','){
      ret.push_back(mp(getid(tokens[i]),getid(tokens[j])+1));
      i = j+1;
    }
  ret.push_back(mp(getid(tokens[i]), getid(tokens.back())+1));
  return ret;
```

```
}

int main(){

  int n=getint(), ttt=getint();
  set< pair<int,int> >::iterator it;
  for(int i=0; i<n; ++i){
    string s;
    cin >> s;
    getid(s);
  }
  for(int tt=0; tt<ttt; ++tt){

    set< pair<int,int> > ranges; // all of these ranges will have x inclusive, and y exclusive
    string s;
    while( (cin >> s), s[0]!='E' ){
      vector< pair<int,int> > a = parse();
      // for all ranges on the line we just read
      for(int i=0; i<sz(a); ++i)
        if(s[0]=='R'){
          // if there is some range that starts before me, fix it
          it=ranges.lower_bound(mp(a[i].x,-1));
          if(it != ranges.begin()){
            --it;
            // if its end goes further than my end, split it in half
            if(it->y > a[i].y){
              pair<int,int> a1(it->x, a[i].x), a2(a[i].y, it->y);
              ranges.erase(it);
              ranges.insert(a1);
              ranges.insert(a2);
            // if it ends within me, it loses the part within me
            }else if(it->y > a[i].x){
              pair<int,int> a1(it->x, a[i].x);
              ranges.erase(it);
              ranges.insert(a1);
            }
          }
          // fix ranges that start equal or after I do
          // first delete all those ending within me
          while(true){
            it = ranges.lower_bound(mp(a[i].x,-1));
            if(it == ranges.end())
              break;
            if(it->y <= a[i].y)
              ranges.erase(it);
            else
              break;
          }
          // then if one starts within me but ends outside, trim it
          it = ranges.lower_bound(mp(a[i].x,-1));
          if(it != ranges.end() && it->x < a[i].y){
            pair<int,int> a1(a[i].y, it->y);
            ranges.erase(it);
            ranges.insert(a1);
          }
        }else{ // s[0]=='S' || s[0]=='A'
          // first erase all ranges that lie within me
          while(true){
            it = ranges.lower_bound(mp(a[i].x,-1));
            if(it != ranges.end() && it->y <= a[i].y)
              ranges.erase(it);
            else
              break;
          }
          // if there is a range that dominates me, this add does nothing
          it = ranges.lower_bound(mp(a[i].x,sz(words)+1));
          if(it != ranges.begin()){
            --it;
            if(it->x <= a[i].x && it->y >= a[i].y)
              goto skipThisAdd;
          }
          // if there is a range before me that ends within me, merge him into me
          it = ranges.lower_bound(mp(a[i].x,-1));
          if(it != ranges.begin()){
            --it;
            if(it->y >= a[i].x){
              a[i].x = it->x;
              ranges.erase(it);
            }
          }
          // if there is a range start equal or after my beginning, but before or equal my end, merge
          it = ranges.lower_bound(mp(a[i].x,-1));
          if(it != ranges.end() && it->x <= a[i].y){
            a[i].y = it->y;
            ranges.erase(it);
          }
          // now insert me
          ranges.insert(a[i]);
skipThisAdd:
          (void)0;
        }
    }

    // the dictionary is constructed, print the answer
    for(it=ranges.begin(); it!=ranges.end(); ++it){
      if(it != ranges.begin())
        printf(", ");
      printf("%s", words[it->x].c_str());
      if(it->y != it->x+1)
        printf(" - %s", words[it->y-1].c_str());
    }
    printf("\n");

  }
  return 0;
}
```

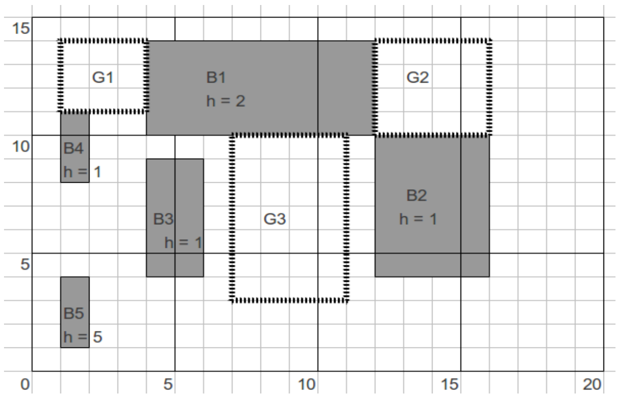
Problem L
Solar Powered Ants

Time Limit: 10 seconds

In Blockland all houses have a rectangular base, vertical walls and a flat roof. Houses, and of course gardens, are all axis-aligned. In addition, local council regulations require that the height of a house be a fixed multiple of the height of the council one-storey building. Blockland is located on the equator of a planet on which the sun always goes from east to west, passing directly overhead at midday. Since there is no seasonal variation, Blocklanders measure time during the day by the length of shadows cast by a one-storey house. They call it shadow time, abbreviated as *ST*. Negative numbers denote shadows to the west (during the morning) and positive numbers denote shadows to the east (during the afternoon). Midday is 0 ST (i.e. zero shadow time).

Once a year, Blockland is invaded from all sides by an army of ants that devour any garden produce they can reach. The ants derive their energy directly from the sun and stop dead if any part of their solar absorbing body is not in direct sunlight. When the arrival hour of the ants is known, Blocklanders need to identify the status of their garden areas as safe or vulnerable to attack. Gardens whose entire areas are not reachable by the ants, due to presence of buildings and/or shadows, are safe and in no need of further protection.

At 2 ST, G1 is sunlit and exposed to attack from the top and the left. G2 is entirely in shadow from building B1 (because that building has a height of 2 and hence shadows of length 4) and therefore safe. G3 is partially sunlit but is protected by buildings B1, B2 and B3 and the shadows of buildings B4 and B5.



Your task is to write a program for Blocklanders that identifies the status of the garden regions.

Input

The input consists of a number of test cases. The description for each test case starts with three integers, *b*, *g*, and *t*, separated by single spaces on a line by themselves. *b*, *g*, and *t* are the number of buildings, the number of gardens and the shadow time of the ants invasion, respectively.
$0 \leq b, g \leq 3000$, and $-1{,}000{,}000 \leq t \leq 1{,}000{,}000$.

Each of the following *b* lines contain 5 integers, *x*, *y*, *w*, *b*, *h*, separated by single spaces. *x* and y represent the coordinates of the bottom left corner of a building, *w* and *b* represent the *x*-span and *y*-span of the building, and *h* represents the height of the building in storeys. Buildings do not overlap other buildings. $0 \leq x, y \leq 1{,}000{,}000$, and $1 \leq w, b, h \leq 1{,}000{,}000$.

Each of the following *g* lines contain 4 integers, *x*, *y*, *w*, *b*, separated by single spaces. *x* and y represent the coordinates of the bottom left corner of a garden, *w* and *b* represent the *x*-span and *y*-span of the garden. Gardens do not overlap buildings or other gardens. $0 \leq x, y \leq 1{,}000{,}000$ and
$1 \leq w, b \leq 1{,}000{,}000$.

Three zeros, separated by single spaces, on a line by themselves indicate the end of data and should not be processed.

Output

For each test case, the output starts with a line that begins with the word "Test" followed by a space and then an integer *n*. *n* is the number of the test case starting with the value "1".

Each of the following *g* lines begins with an integer, which is a garden number followed, after a blank space, by the assessment of the whole garden area as "safe" or "vulnerable". Gardens are assigned numbers, starting with the value "1", according to the order of their appearance in the input data, and must appear in that order in the output.

Sample Input	Output for the Sample Input

5 3 2
4 10 8 4 2
12 4 4 6 1
4 4 2 5 1
1 8 1 3 1
1 1 1 3 5
1 11 3 3
12 10 4 4
7 3 4 7
1 3 1
1000 2000 1 1 1000
2000 2000 1 1
2001 2000 1 1
2000 1999 1 1
0 0 0
	Test 1
	1 vulnerable
	2 safe
	3 safe
	Test 2
	1 safe
	2 vulnerable
	3 vulnerable

L: Solar Powered Ants

First observe that there are at most (3000 gardens + 3000 buildings) $*2$ coordinates in each x and y. So compress the coordinates and now all the coordinates are in the range $0 \ldots 11,999$.

So let $D = 12,000$ be the maximum possible dimension of the board.

I cannot just iterate over all gardens and mark all their cells as shadowed, as some cells would get marked many times and the solution would time out.

I will use a sweepline to mark cells as shadowed.

For a tutorial on sweepline I direct you to topcoder:

`http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lineSweep`

The sweepline works by moving over all x-coordinates, and maintaining a set of all left edges of shadows that have opened up, but not yet closed. Then we iterate over the list of open shadows to figure out for which y-coordinates there is a cell in shadow with this x-coordinate.

Now that I have the shadows marked, I can flood fill to find the squares reachable by ants in time linear in the number of cells. This is just a DFS from all edges of the board, never visiting a square twice, nor visiting a square that is in shadow.

Then for each cell (x, y) I compute the prefix sum of reachable cells (in this case I compute suffix sum), the total number of reachable cells (x', y') such that $x' \geq x$ and $y' \geq y$. If a is the orginal array, and we want to compute the *sum* array, then this can be done using dynamic programming:

$$sum_{x,y} = sum_{x+1,y} + sum_{x,y+1} - sum_{x+1,y+1} + a_{x,y}$$

And now I query for each garden whether it has a reachable cell in constant time using

$$\sum_{x_{lo} \leq x < x_{hi}, y_{lo} \leq y < y_{hi}} a_{x,y} = sum_{x_{lo},y_{lo}} - sum_{x_{hi},y_{lo}} - sum_{x_{lo},y_{hi}} + sum_{x_{hi},y_{hi}}$$

The time complexity of this solution is $\mathcal{O}(D^2)$ per test case. All the coordinate compress stuff is $\mathcal{O}(D \lg D)$. Marking shadows, flood-filling reach, and computing prefix sum are all $\mathcal{O}(D^2)$. The final iterate over all gardens to test if they contain any reachable cells is $\mathcal{O}(G)$, the number of gardens, as each query is $\mathcal{O}(1)$.

```
#include<bits/stdc++.h>
using namespace std;
#define mp make_pair
#define pb push_back
#define sz(x) ((int)(x).size())
#define x first
#define y second
typedef long long ll;
inline int getint() { int a; scanf("%d", &a); return a; }

typedef pair<int,int> ii; // x,y coordinates of a point
vector<int> xs, ys;
const int N = 1<<14;
char shadow[N][N];
int reach[N][N];

int onboard(int i, int j){ return i>=0 && j>=0 && i<sz(xs) && j<sz(ys); }
int di[4] = {-1, 0, 1, 0};
int dj[4] = {0, 1, 0, -1};
void dfs(int i, int j){
  reach[i][j] = 1;
  for(int k=0; k<4; ++k){
    int ni=i+di[k], nj=j+dj[k];
    if(onboard(ni, nj) && !shadow[ni][nj] && !reach[ni][nj])
      dfs(ni, nj);
  }
}

int main(){
  const ll MAX_COORD = 2e6 + 10;
  int nb, ng, tt=0;
  ll t; // don't want t*h to overflow
  while( (nb=getint(), ng=getint(), t=getint()), nb || ng || t ){
    printf("Test %d\n", ++tt);

    // read in all buildings and gardens
    // actually we will store the shadow of the building as that is all we care about
    vector< pair<ii,ii> > b, g;
    for(int i=0; i<nb; ++i){
      int x=getint(), y=getint(), w=getint(), br=getint(), h=getint();
      b.pb(mp(ii(x,y),ii(x+w,y+br)));
      if(t > 0)
        b[i].y.x = min(MAX_COORD, b[i].y.x + h*t);
      else
        b[i].x.x = max(-MAX_COORD, b[i].x.x + h*t); // t is negative, so the x coordinate will move back to the left appropriate
    }
    for(int i=0; i<ng; ++i){
      int x=getint(), y=getint(), w=getint(), br=getint();
      g.pb(mp(ii(x,y),ii(x+w,y+br)));
    }

    // coordinate compression - remap all points into the range 0 ... num distinct coordinates - 1
    xs = vector<int>();
    ys = vector<int>();
    for(int i=0; i<sz(b); ++i){
      xs.pb(b[i].x.x);
      ys.pb(b[i].x.y);
      xs.pb(b[i].y.x);
      ys.pb(b[i].y.y);
    }
    for(int i=0; i<sz(g); ++i){
      xs.pb(g[i].x.x);
      ys.pb(g[i].x.y);
      xs.pb(g[i].y.x);
      ys.pb(g[i].y.y);
    }
    sort(xs.begin(), xs.end());
    xs.resize(unique(xs.begin(), xs.end())-xs.begin());
    sort(ys.begin(), ys.end());
    ys.resize(unique(ys.begin(), ys.end())-ys.begin());
    for(int i=0; i<sz(b); ++i){
      b[i].x.x = lower_bound(xs.begin(), xs.end(), b[i].x.x)-xs.begin();
      b[i].x.y = lower_bound(ys.begin(), ys.end(), b[i].x.y)-ys.begin();
      b[i].y.x = lower_bound(xs.begin(), xs.end(), b[i].y.x)-xs.begin();
      b[i].y.y = lower_bound(ys.begin(), ys.end(), b[i].y.y)-ys.begin();
    }
    for(int i=0; i<sz(g); ++i){
```

```
  g[i].x.x = lower_bound(xs.begin(), xs.end(), g[i].x.x)-xs.begin();
  g[i].x.y = lower_bound(ys.begin(), ys.end(), g[i].x.y)-ys.begin();
  g[i].y.x = lower_bound(xs.begin(), xs.end(), g[i].y.x)-xs.begin();
  g[i].y.y = lower_bound(ys.begin(), ys.end(), g[i].y.y)-ys.begin();
}

// clear global data between each run
for(int i=0; i<sz(xs); ++i)
  for(int j=0; j<sz(xs); ++j)
    shadow[i][j] = reach[i][j] = 0;

// sweep line to find all cells in shadow
// < <time of event, add/remove>, < range of cells covered > >
// time of event is the x-coordinate, the range is the y-range
// add/remove is whether the range covered is being turned on or off
vector< pair<ii,ii> > event;
for(int i=0; i<sz(b); ++i){
  event.pb(mp(ii(b[i].x.x,1),ii(b[i].x.y,b[i].y.y)));
  event.pb(mp(ii(b[i].y.x,0),ii(b[i].x.y,b[i].y.y)));
}
sort(event.begin(),event.end());
// contains all ranges presently covered
multiset<ii> cover;
int upto = 0;
for(int x=0; x<sz(xs); ++x){
  // process all relevant events so we have the right range covered for this time
  for( ; upto < sz(event) && event[upto].x.x == x; ++upto)
    if(event[upto].x.y)
      cover.insert(event[upto].y);
    else
      cover.erase(cover.lower_bound(event[upto].y));
  // mark the shadows for this x-coordinate
  int y = 0;
  for(multiset<ii>::iterator it=cover.begin(); it!=cover.end(); ++it)
    // N.B. y must start at max(y, it->x), if you just restart at it->x you could mark cells
    // multiple times and time out
    for(y = max(y, it->x); y < it->y; ++y)
      shadow[x][y] = 1;
}

// now flood-fill to see what is reachable
// dfs from all edges of the board
for(int i=0; i<sz(xs); ++i){
  if(!shadow[i][0] && !reach[i][0])
    dfs(i, 0);
  if(!shadow[i][sz(ys)-1] && !reach[i][sz(ys)-1])
    dfs(i, sz(ys)-1);
}
for(int i=0; i<sz(ys); ++i){
  if(!shadow[0][i] && !reach[0][i])
    dfs(0, i);
  if(!shadow[sz(xs)-1][i] && !reach[sz(xs)-1][i])
    dfs(sz(xs)-1, i);
}

// compute the prefix sum such that we have the sum values of all points with greater or equal x and y
// base case - handle the edges
for(int i=sz(xs)-2; i>=0; --i)
  reach[i][sz(ys)-1] += reach[i-1][sz(ys)-1];
for(int j=sz(ys)-2; j>=0; --j)
```

```
      reach[sz(xs)-1][j] += reach[sz(xs)-1][j-1];
    // general case - handle the body
    for(int i=sz(xs)-2; i>=0; --i)
      for(int j=sz(ys)-2; j>=0; --j)
        reach[i][j] += reach[i+1][j] + reach[i][j+1] - reach[i+1][j+1];

    // test all gardens for being reachable by the ants
    for(int i=0; i<sz(g); ++i){
      // the number of reachable compressed squares within the garden reachable by the ants
      int reachable = reach[g[i].x.x][g[i].x.y]
                    - reach[g[i].y.x][g[i].x.y]
                    - reach[g[i].x.x][g[i].y.y]
                    + reach[g[i].y.x][g[i].y.y]
                    ;
      // we are vulnerable if so much as 1 square is reachable
      printf("%d %s\n", i+1, reachable ? "vulnerable" : "safe");
    }

  }
  return 0;
}
```