

bubble
BUBBLECUP.ORG **cup**



STUDENT
CODING
COMPETITION

PROBLEM SET BOOKLET



Microsoft
Development Center Serbia

BUBBLE CUP 2013

Student programming contest
Microsoft Development Center Serbia

Problem set & Analysis from the Finals and Qualification rounds

Belgrade, 2013

Scientific committee:

Andreja Ilić
Andrija Jovanović
Marko Živanović
Mladen Radojević
Dimitije Filipović
Dejan Kraković

Qualification analyses:

Aleksandar Ivanović
Aleksandar Milovanović
Marko Rakita
Karolis Kusas
Petar Velicković
Dusan Zdravković
Nenad Bozidarević
Boris Grubić
Vanja Petrović Tanković
Vladislav Haralampiev
Marko Stanković
Nikola Milosavljević
Demjan Grubić
Nikola Obradović
Dimitrije Dimić
Nikola Stojiljković

Cover:

Sava Čajetinac

Typesetting:

Boris Grubić

Proofreader:

Andrija Jovanović

Volume editor:

Dragan Tomić

Contents

Preface.....	5
About Bubble Cup and MDCS.....	6
Bubble Cup High School Finals 2013.....	7
Problem A: Game	9
Problem B: Turing.....	12
Problem C: Code.....	15
Problem D: Jumping	18
Problem E: Hacker	21
Problem F: Scammer	25
Problem G: Unary	28
Problem H: Graffiti	30
Qualifications	32
Problem R1 01: A Famous Grid (ID: 11582).....	33
Problem R1 02: Projections Of A Polygon (ID: 1431)	35
Problem R1 03: Circular game (ID: 4309)	37
Problem R1 04: Two Famous Companies (ID: 11579)	40
Problem R1 05: Special Graph (ID: 13529)	43
Problem R1 06: Dinosaur Menace (ID: 7187)	45
Problem R1 07: The Ball (ID: 8391)	47
Problem R1 08: Palindromes (ID: 9748)	50
Problem R1 09: Shrinking Polygons (ID: 3415)	52
Problem R1 10: [CH] Tetris AI (ID: 758)	54
Problem R2 01: Avoiding SOS Grids (ID: 6999)	58
Problem R2 02: High and Low (ID: 12210)	60
Problem R2 04: Tower Game (Hard) (ID: 7857)	62
Problem R2 05: Fibonacci recursive sequences (hard) (ID: 12009)	68
Problem R2 06: Help Blue Mary Please! (Act I) (ID: 1457)	74
Problem R2 07: Problems Collection (Volume X) (ID: 1815)	76
Problem R2 08: AB-words (ID: 177)	79
Problem R2 09: Santa Claus and the Presents (ID: 240).....	81
Problem R2 10: Robo Eye (ID: 2629)	84

Preface

Dear Finalist of Bubble Cup 6,

Thank you for participating in the sixth edition of the Bubble Cup. On behalf of Microsoft Development Center Serbia (MDCS), I wish you a warm welcome to Belgrade and I hope that you will enjoy yourself.

MDCS has a keen interest in putting together this world class event. Most of our team members participated in similar competitions in the past and have passion for solving difficult technical problems.

This edition of the Bubble Cup is special. Current (sixth) edition of the Bubble Cup is the most international competition that we have had so far. For the first time we have two categories. University students will be battling with problems in a 24 hour contest. High school students will continue competing in our traditional format. Not only do we have the participants from the region (Bulgaria, Croatia, Montenegro, Serbia) but also teams from Germany, Lithuania and Poland fought their way to the Finals. This means that the Bubble Cup has increased its scope and popularity every year in its history.

Given that we live in the world where technological innovation will shape coming decades, your potential future impact on humankind will be great. Take this opportunity to advance your technical knowledge and to build relationships that could last you a lifetime.

I wish you all warm welcome to Belgrade and Serbia.

Thanks,

Dragan Tomić

MDCS Group Manager/Director

About Bubble Cup and MDCS

Bubble Cup is a coding contest started by Microsoft Development Center Serbia in 2008 with a purpose of creating a local competition similar to the ACM Collegiate Contest, but soon that idea was outgrown and the vision was expanded to attracting talented programmers from the entire region and promoting the values of communication, companionship and teamwork.

The contest has been growing in popularity with each new iteration. In its first year close to 100 participants took part and this year, 2013, it attracted more than 500 participants.

This year the emphasis was on keeping intact all of the things that made Bubble Cup work in previous years but taking every opportunity to tweak and subtly improve the format of the contest. For the first time we have two categories. University students will be battling with problems in a 24 hour contest. High school students will continue competing in our traditional format.

Microsoft Development Center Serbia (MDCS) was created with a mission to take an active part in conception of novel Microsoft technologies by hiring unique local talent from Serbia and the region. Our teams contribute components to some of Microsoft's premier and most innovative products such as SQL Server, Office & Bing. The whole effort started in 2005, and during the last 8 years a number of products came out as a result of great team work and effort.

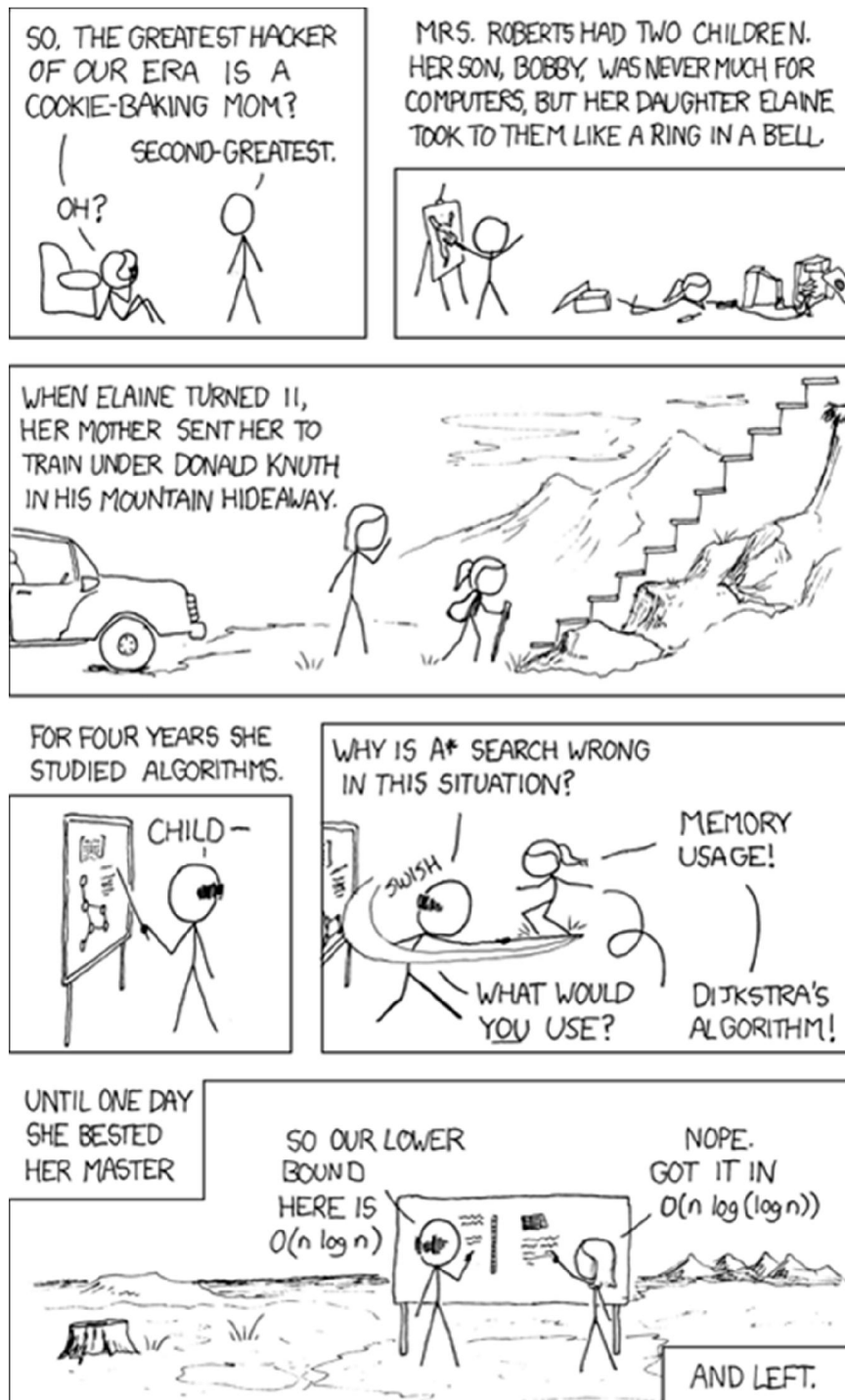
Our development center is becoming widely recognized across Microsoft as a center of excellence for the following domains: computational algebra engines, pattern recognition, object classification, computational geometry and core database systems. The common theme uniting all of the efforts within the development center is applied mathematics. MDCS teams maintain collaboration with engineers from various Microsoft development centers around the world (Redmond, Israel, India, Ireland and China), and Microsoft researchers from Redmond, Cambridge and Asia.



Microsoft[®]
Development Center Serbia

Bubble Cup High School Finals 2013

Problem set & Analysis



The finals of Bubble Cup 6 for high school students were held on September 7th 2013 in Microsoft Development Center Serbia. There were 13 teams in the finals, trying to tackle 8 problems in five hours.

The battle was really tight. Some of the tasks proved to be too hard for the competitors, as three problems remained unsolved. Every team managed to solve at least two problems, with six teams solving five problems. At the end, penalty was the deciding factor for top places. Me[N]talci were the most efficient, winning the top spot, followed by We're not on IOI, so we are here and abbabaab.

Place	Team	Score	Penalty	Total submissions	Accepted submissions	Rejected submissions
1.	Me[N]talci	5	458	7	5	2
2.	We're not on IOI, so we are here	5	674	10	5	5
3.	abbabaab	5	690	16	5	11
4.	2bubbles1cup	5	734	10	5	5
5.	Robimy pompki z klaśnięciem za plecami	5	747	13	5	8
6.	ksiwlo	5	896	15	5	10
7.	μwaveZ~	4	679	4	4	0
8.	bubble crew	4	783	18	4	14
9.	Svak' Smeta	3	190	3	3	0
10.	The Mongooses	2	140	4	2	2
11.	Gimnazija Sombor	2	182	3	2	1
12.	EatSleepCode	2	248	4	2	2
13.	The C's	2	378	4	2	2

Table 1. Detailed results and submission statistics per team

Problem A: Game

Authors

Dušan Zdravković

Implementation and analysis

Dušan Zdravković

Vanja Petrović Tanković

It is given a tree with N nodes and number K . Lou is playing a game on that tree in such way:

- In one move, he chooses one node and some K neighbors of that node (he can choose only node with at least K neighbors) and destroys them all. Destroyed nodes can't be used later.

Since he wants that his game lasts as long as possible, tell him what is the maximal number of nodes he can destroy.

Input

The first line contains two integer numbers N and K . Each of the next $N - 1$ lines contains two integer numbers A and B (in the range $1..N$) which represents that there is the edge in the tree between nodes A and B .

Output

Output should contain one integer number which represents the maximal number of nodes that Lou can destroy.

Constraints

- $1 \leq N \leq 10^5$
- $0 \leq K \leq 10^5$
- $1 \leq A, B \leq N$

Example input

```
9 2
1 2
1 3
2 4
2 5
3 6
3 7
4 8
4 9
```

Example output

```
9
```

Example explanation

In the first move, he can choose node 4 and his neighbors 8 and 9 and destroy them all. In the second move he can choose node 3 and his neighbors 6 and 7, and finally, he can choose node 2 and nodes 1 and 5.

Solution and analysis:

The task is to find how many disjunctive (K+1) stars there are in a tree.

Usually, one of the first things that come to the mind in the problem like this is dynamic programming in a tree, which leads to the solution in this task as well.

For each vertex (node) x, we will calculate the maximum number of disjunctive stars that we can make in its subtree. To do that, we will need several values:

d[x].Zero – the maximum number of stars that we can make if we do not take vertex x at all

d[x].Kdown – the maximum number of stars that we can make if we take vertex x and its K children

d[x].Onedown – the maximum number of stars that we can make if we take vertex, one of his children and K-2 children of chosen children

d[x].Kminus1down – the maximum number of stars that we can make if we take vertex x, his father and his K-1 children

d[x].MaxWithoutFather – max(d[x].Zero, d[x].Kdown, d[x].Onedown).

We will calculate these values in a following way:

For d[x].Zero is easy, just sum up all d[u].MaxWithoutFather for all u where u is child of x.

It is a bit harder for d[x].Kdown and d[x].Kminus1down. First, we will show how to calculate d[x].Kdown and then d[x].Kminus1down can be computed in a similar way.

Suppose that the set S is a set of K vertices that we take to form a star with node x (their father). The number of stars in a subtree of node x will then be:

In all formulas below, u is every child of x

$$\sum_{u \in S} d[u].Zero + \sum_{u \notin S} d[u].MaxWithoutFather$$

Therefore, we want to find the set S for which that sum is maximal. The sum above can be written in a following way as well:

$$\sum_u d[u].MaxWithoutFather - \sum_{u \in S} (d[u].MaxWithoutFather - d[u].Zero)$$

Because first sum is constant for all sets S, we should just choose K children (u) for which $d[u].MaxWithoutFather - d[u].Zero$ are minimal. We can do that simply by sorting children by that value. For d[x].Kminus1down it's all the same, just instead of K children, we will choose first K-1 children.

For $d[x].\text{Onedown}$, we should choose node y for which this sum is maximal:

$$\sum_u d[u].\text{MaxWithoutFather} - d[y].\text{MaxWithoutFather} + d[y].\text{Kminus1down}$$

And that is the node with maximal $d[y].\text{Kminus1down} - d[y].\text{MaxWithoutFather}$.

After all these values calculated, the answer is simple: $d[\text{root}].\text{MaxWithoutFather}$.

Problem B: Turing

Authors

Vanja Petrović Tanković

Implementation and analysis

Marko Živanović

Vanja Petrović Tanković

You are given a Turing machine. Turing machine in this task consists of

1. A **state** register that stores the state of Turing machine. There are 52 different states, labeled with lowercase and uppercase letters of English alphabet ('a' – 'z' and 'A' – 'Z'). At the beginning, the machine is in the state 'a'. State 'F' is the final state – the execution of the machine halts if the machine gets into state 'F'.
2. A **tape** divided into 16 cells, one next to the other (a row of cells). Each cell contains either a symbol 0 or a symbol 1. At the beginning, symbol of each cell is set to 0.
3. A **head** that can read and write symbols on the tape. The head is a pointer to one cell. The head can move to adjacent cells (left or right), but it must not move outside of tape bounds. At the beginning, the head points to the first (leftmost) cell.
4. A **program** for the machine – set of instructions (transition functions). Instruction is a 5-tuple: $Q_c b_c \rightarrow Q_n b_n D$ that, given the $state(Q_c)$ the machine is currently in and the $symbol(b_c)$ it is currently reading (symbol of a cell which the head is currently pointing to) tells the machine to do the following in sequence:
 - a. Write the new $symbol(b_n)$ to a cell which the head is currently pointing to
 - b. Move the head, which is described by $direction(D)$ ('L' – left; 'R' – right; 'N' – stay at the same cell)
 - c. Change the state of the machine to the new $state(Q_n)$

One operation is one execution of the instruction. Your task is to write a valid program for this Turing machine that does at least 1024 operations.

Program is valid if it halts (reaches the final state), head does not move outside of tape bounds, instruction set does not contain duplicate $Q_c b_c$ pairs and machine never reaches a pair $Q_c b_c$ such that no instruction is defined for the pair (unless Q_c is the final state).

Input

There is no input.

Output

The first line of output should contain one positive integer N – the number of instructions. Each of the next N lines should contain 5 characters separated by spaces representing one instruction – $V W X Y Z$

V represents Q_c ; W represents b_c ; X represents Q_n ; Y represents b_n ; Z represents D ;

$V, X \in \{'a', \dots, 'z', 'A', \dots, 'Z'\}$;

$W, Y \in \{0, 1\}$;

$Z \in \{L, R, N\}$

Turing machine was invented in 1936 by Alan Turing. It is a hypothetical device representing a computing machine. Turing machine is important in computational complexity theory, because it is easy to analyze mathematically and it is believed it is as powerful as any other model of computation. The Church-Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine. There are different types of Turing machines, and some are used to define complexity classes, such as deterministic Turing machines, non-deterministic Turing machines, probabilistic Turing machines etc. However, Turing machine in this task is not a real Turing machine, because it does not have infinite resources like the usual Turing machine. The tape is not unlimited and there is only 52 possible states of the machine.

This solution to this task is intended to be made “by hand”. While it should be possible to write a program that will write a Turing machine program, it is much easier to use our human intelligence and intuition to come up with the solution to this task. There are several different ideas for programs that would execute at least 1024 operations. We are going to present one that is rather simple, has potential to do a lot more than 1024 operations and does not need a lot of instructions.

Since the cells of the tape can have either symbol 0 or symbol 1, we can view the tape as a binary number, which is the core of the idea. By counting numbers on the tape (incrementing by one), we can count 2^{16} numbers, since the tape has 16 cells. This alone is greater than 1024 operations, and we have not even taken into account the operations needed to increment each number. Of course, it might not be possible to count all the 2^{16} numbers because we have to stop somehow, but it is not needed.

Our program could count up to $2^9 - 1$, which should be more than enough when we take into account the operations to increment each numbers. There are several different ways to achieve this. The more significant bits will be to the right on the tape, so the least significant bit of the number will be the leftmost bit. It is possible to use the leftmost cell as the least significant, but it is better to use it for something else so we can shorten our program.

At the beginning, we will set the value of the leftmost cell to 1. This will be an indicator that it is the end of the tape. It will be used for the loop, which is going to be explained later on. We are going to use a different state for each cell when going to the right. So, for the first bit we will use $state_1$, for the second $state_2$, etc. (states can be mapped to English alphabet letters at the end). The most significant bit is marked with $state_9$ (because we are using 9 cells to represent a number). When incrementing a number, we start incrementing from the lowest bit.

If the bit is incremented from 1 to 0, we carry one to the next bit to the right. When moving the head to the cell right, we go from the $state_i$ to $state_{i+1}$. The special case is when we reach $state_9$ and increment from 1 to 0 and it means that we have counted all the numbers and can enter the final state and finish the execution.

If the current bit is incremented from 0 to 1, it means that we do not have to carry one and we have finished the incrementation phase. Now we have to return to the least significant bit. This is a part where the 1 in the leftmost cell comes in handy. All the cells left from the one the head is currently pointing at will have a value 0, except the one in the leftmost cell. We can write a simple loop that will move us to the leftmost cell, and then we can just use an additional instruction to move to the cell for the least significant bit (one to the right).

The loop is simple – while there is a 0 in the current cell, stay in the same state and move to the left. The loop ends when we read the symbol 1. We then change the state and move to the right (lowest bit).

Example of this Turing machine loop:


```
X 0 X 0 L
X 1 Y 1 R
```

When we reach the least significant bit, we can start with the incrementation phase again. This program needs only about 10 different machine states and it is quite short. The full program is given below:

```
21
a 0 x 1 R
x 0 x 1 N
x 1 b 0 R
b 0 L 1 L
b 1 c 0 R
c 0 L 1 L
c 1 d 0 R
d 0 L 1 L
d 1 e 0 R
e 0 L 1 L
e 1 f 0 R
f 0 L 1 L
f 1 g 0 R
g 0 L 1 L
g 1 h 0 R
h 0 L 1 L
h 1 i 0 R
i 0 L 1 L
i 1 F 0 R
L 0 L 0 L
L 1 x 1 R
```

Problem C: Code

Authors

Dušan Zdravković

Implementation and analysis

Dušan Zdravković

Boris Grubić

Gennady just passed lection Recursion on BubbleBee site, and then, he easily solved one task writing just a few lines of code:

```
int_64 a[N];

int_64 sum(int x, int y) {
    int_64 s;
    s = 0;
    for(int i = x; i <= y; i++) s+=a[i];
    return s;
}

int_64 solve(int_64 left, int_64 right, int_64 index) {
    int_64 tmp,res;
    if (left == right) return a[left];
    res = solve(left, index, left) + sum(left, index) + solve(index+1, right, index+1) + sum(index+1, right);
    if (index+1 < right) {
        tmp = solve(left, right, index+1);
        if (tmp < res) res = tmp;
    }
    return res;
}

int_64 get_Answer() {
    read N;
    read array A of N elements;
    return solve(1, N, 1);
}
```

Unfortunately, he hasn't learned about complexity of algorithms yet, so he didn't know that his code is very inefficient. Please, help him to efficiently solve this task for all test cases.

Input

The first line contains one integer N . Next line contains N integers that represents array A .

Output

Output the same number that function `get_Answer()` from Gennady's code returns.

Constraints

- $1 \leq N \leq 3000$
- $0 \leq A_i \leq 10^9$

Example input

5
3 5 1 2 7

Example output

57

Solution and analysis:

First we should notice that the function `solve()` calculates the solution for an interval $[left, right]$, while the `index` is used to iterate through that interval as a loop. Then, we could make an equivalent `solve2()` function:

```
int_64 solve2(int_64 left, int_64 right) {
    int_64 res;
    if (left == right) return a[left];
    res = infinity;
    for(int index = left+1; index < right; index++) {
        res = min(res, solve2(left, index) + sum(left, index) + solve2(index+1,
right) + sum(index+1, right));
    }
    return res;
}
```

Now we can use a matrix $d(N \times N)$ to store the solution for each $[left, right]$ interval, and not make further recursive calls (this method is also known as *memoization*). The code complexity would then be $O(N^3)$, as the `sum()` function could be calculated in $O(1)$ in a well-known way, storing prefix sums.

As $O(N^3)$ is not good enough, we should further optimize the code.

Let `mid_point[$left, right$]` be the index for which the solution of function `solve2($left, right$)` has been found. Then, it can be proven that the inequalities `mid_point[$left, right$] \geq mid_point[$left, right - 1$]` and `mid_point[$left + 1, right$] \leq mid_point[$left, right$]` are always valid. This is also known as the *Knuth optimization*¹.

Now, if we keep track of `mid_point` for each interval in a separate matrix, we could adjust the range in the `solve2()` function loop in which we search the `mid_point[$left, right$]` index, as shown in the following equivalent `solve3()` function:

¹ *The Art of Computer Programming*, Donald Knuth

```

int_64 d[N][N] = {infinity}, mid_point[N][N];
int_64 solve3(int_64 left, int_64 right) {
    int_64 res = infinity;
    if (d[left, right] != infinity) return d[left, right];
    if (left == right) {
        mid_point[left, left] = left;
        d[left, left] = a[left];
        return d[left, left];
    }
    for(int index = mid_point[left, right-1]; index < mid_point[left+1, right]; index++) {
        tmp = solve3(left, index) + sum(left, index) + solve3(index+1, right) + sum(index+1, right);
        if (tmp < res) {
            mid_point[left, right] = index;
            res = tmp;
        }
    }
    d[left,right] = res;
    return res;
}

```

As each element is checked exactly once while increasing the right interval limit, as well as exactly once while increasing the left interval limit, the overall complexity is $O(N^2)$. The exact mathematical proof of this is left to the reader to devise. 😊

Problem D: Jumping

Authors

Dušan Zdravković

Implementation and analysis

Dušan Zdravković

Andrija Jovanović

Egor has written an array which consists of N natural numbers and in which all numbers are smaller than or equal to N . Now he is playing with it in a following way:

- In the beginning, he circles the first number in the array.
- Whenever he circles a number X , he moves to the X th element in the array, circles it and repeats the procedure.

Egor always makes exactly 94294032599379535 steps and then stops playing. However, he has noticed that, sometimes when he finishes the game, some elements of the array have never been circled, and he does not like that. Because of that, he wants to change some elements of the array before he starts playing in such a way that, in the end, every number has been circled at least once. Help him to calculate the minimum number of elements that he needs to change in order to reach his aim.

Input

The first line contains one integer N number of elements of the array. Second line contains N integers – elements of the array.

Output

Output contains one integer representing the minimum number of elements that Egor needs to change in order to reach his goal – to circle every element in the array.

Constraints

- $1 \leq N \leq 10^6$
- $1 \leq A_i \leq N$

Example input

```
5
2 3 2 1 4
```

Example output

```
1
```

Example explanation

If Egor doesn't change any of the elements, then he will first circle the first number (2), then the second number (3), then the third number (2), then the second number (3), then the third number... and in the end only the first, the second and the third number will be circled.

If he changes the third number into 5, all elements will be circled.

Solution and analysis:

This problem is not difficult, but there is a number of edge cases we have to be careful about.

First, let's rephrase the problem statement in terms of graph theory. We have a directed graph with N vertices, and with exactly one edge going out of each vertex. Our goal is to transform it into a graph from

which, starting at node 1, we can traverse the entire graph – and we have to do it with the minimal number of changed edges. In most cases this will have to be a full cycle. The only other option is a path starting from vertex 1, traversing the entire graph but not looping back into vertex 1.

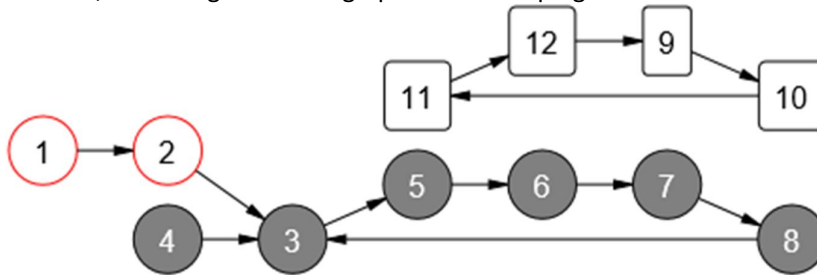


Figure 1. An example of dividing the graph into three types of shapes – a line (1, 2), tail (3-8) and cycle (9-12)

What is the shape of our original graph? It is easy to see that the weakly connected components of the graph are cycles, some of which have trees attached to them (like vertices 1, 2 and 4 in Figure 1). We will split each of these graph-tree components into “tail” and “line” shapes, by picking a vertex with an in-degree of 0 and following the path from it until we close a cycle or hit a vertex that we have already assigned to another line or tail.

Obviously, the split into lines, tails and cycles will in general not be uniquely determined for a graph, but the total number of shapes will. This is because each weakly connected component will contain either exactly one shape (if it is a cycle) as many shapes as it has vertices with an in-degree of zero (if it is not). We will denote this number with K .

Let us first concentrate on the case where we construct a full cycle. We can claim the following:

Lemma 1.

If the graph is not already a full cycle, the number of alterations needed to make it a full cycle cannot be less than K - the number of shapes in the graph.

Outline of proof.

If the graph consists of just a single tail shape, we obviously need at least one alteration. Otherwise, we can notice that none of the shapes contain the entire set of vertices. This means that each shape will need at least one edge which connects it to the other shapes, so it cannot remain unaltered.

Now let's construct an algorithm that makes exactly K alterations. For each shape we will select a start vertex and an end vertex. For tails and lines there is a unique pick, while for cycles we can pick a vertex at random and make it both the start and the end. We then apply the following algorithm:

1. Initialize a set of selected shapes $S = \{\emptyset\}$
2. Pick a shape s and add it to S
3. Select a shape $t \notin S$, and alter the edge going out of the end vertex of s so that it points to the beginning vertex of t
 - If there is no such shape t , connect the end vertex of s to the beginning vertex of the shape which was picked first and terminate the algorithm
4. Repeat the steps 2 and 3 with the shape t in place of s

It is clear that this algorithm ends up in the correct state after K iterations, each iteration altering one edge. So since there is always a way to solve the problem with K changes, and by lemma 1 we know that we can never solve it with less than K , this means that the answer is exactly K .

Here we can notice that the change to the algorithm in order to cover our other end state (a path that begins with vertex 1 but does not necessarily loop back to it in the end) is minimal. Namely, if the vertex 1 was the starting point of a tail or a line, we will pick that shape first, and omit the step in which we connect the end vertex of the last shape to it. And if vertex 1 was part of a cycle, we will pick it for both the start and end vertex, pick that cycle first and proceed as in the previous case.

This can be done in the following manner:

1. First, check if the graph already satisfies the conditions of the problem; if it does, output 0 and terminate.

2. Find all vertices that do not have any incoming edges. Denote their number with B . These vertices are beginnings of tails and lines. If vertex 1 is among them, set a flag F .
3. Mark all vertices that belong to shapes beginning with vertices from the previous step. All unmarked vertices are parts of cycles. If vertex 1 is unmarked, set the F flag.
4. Count the number of cycles among the unmarked vertices. Denote this number with C .
5. If F is not set, output $B + C$ as the solution and terminate. Else, output $B + C - 1$ as the solution and terminate.

Each of the first four steps takes $O(N)$ time – notice that no vertex needs to be visited more than once in any individual step – so the whole solution takes $O(N)$ time as well. The memory complexity is also $O(N)$.

Problem E: Hacker

Authors

Nikola Stojiljković

Implementation and analysis

Mladen Radojević

Nikola Stojiljković

Bob Bubbles and his brother Bub Bubbles always competed with each other. Since Bub Bubbles was making a lot of bubblars (money currency in Bubbleland) with his scamming machine, Bob Bubbles thought about hacking into a bank so he can have more bubblars than his brother.

In order to do that, he has opened N programs on his computer. Each program is a rectangular window with sides parallel to the sides of his high resolution screen. Windows can overlap. He also needed a special program Bubble Tracker to show him how close is he to be caught by cyber police so he could prevent it. The window of Bubble Tracker is a square and Bob wanted it always to be visible (it cannot intersect with other windows and it must be inside the screen, but its sides can lie on the sides of other windows). He also needed it to be as big as possible but he couldn't resize any of the already opened windows because it would lower his performance so he asked you for help. Help Bob Bubbles find the maximal length of the side of Bubble Tracker that can be fit in the screen so he can beat his brother.

Input

The first line of the standard input contains two numbers W and H ($1 \leq H, W \leq 10^9$), representing width and height of the screen respectively. Bottom-left corner of the screen is $(0, 0)$ and top-right corner is (W, H) . Next line contains number N ($1 \leq N \leq 40000$), number of programs Bob Bubbles has opened. Each of the next N lines contains four integers X_i, Y_i, W_i, H_i ($1 \leq X_i, Y_i, W_i, H_i \leq 10^9, X_i + W_i \leq W, Y_i + H_i \leq H$), coordinates of bottom-left corner, width and height of the i_{th} window respectively.

Output

On the first and only line of the standard output print the maximal length of the side of Bubble Tracker window.

Example input

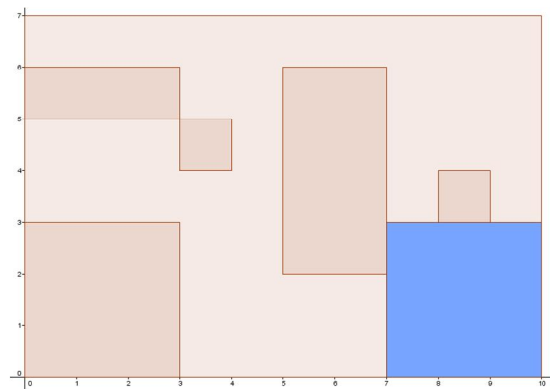
```
10 7
5
0 0 3 3
3 4 1 1
0 5 3 1
5 2 2 4
8 3 1 1
```

Example output

```
3
```

Example explanation

One of the possible solutions of the sample test is shown in the picture with bottom-left corner in (7, 0).



Solution and analysis:

We are given starting rectangle with coordinates $(0, 0, \text{width}, \text{height})$ and N rectangles in it. Our task is to find length of the biggest square that can be placed in the starting rectangle so that it doesn't overlap with any other of the N given rectangles.

Suppose there is only one rectangle $R(0, 0, x, y)$. We are going to resize it by some value k so it becomes $R1(0, 0, x + k, y + k)$. We can notice that for any point A that is not in the resized rectangle $R1$ (it can lie on the edge of rectangle $R1$ because sides of windows in the task can touch each other), we can put square S with side of length k and top-right corner in A so that S doesn't overlap with starting rectangle R . Also, for each point B that is inside rectangle $R1$ we cannot put square S with top-right corner in B because it will overlap with starting rectangle R . For example, if B is in the resized part of the rectangle, either left or bottom edge of S will be inside rectangle R because length of the resized part is k .

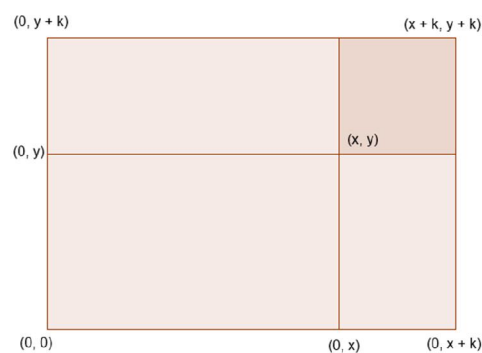


Image 1. Rectangle $(0, 0, x, y)$ and resized rectangle $(0, 0, x + k, y + k)$

If we resize each rectangle by k (if rectangle resized by k goes out from the starting rectangle then it should be trimmed so it stays inside) and there exists point A which is not inside of any resized rectangle, we can put square S with top-right corner in A such that it doesn't overlap with any of the original (non-resized) rectangles. Notice that bottom and left edges of the starting rectangle $(0, 0, 0, \text{height})$ and $(0, 0, \text{width}, 0)$ should be considered as rectangles too. We should only check if some point A exists (we don't need to know its location).

Solution will always be an integer and that comes from the fact that if there exists solution k' for which opposite sides don't lie on two rectangles then there exists some solution $k > k'$ (we can expand square until it touches at least two rectangles). And since all the inputs are integers then the distance between two rectangles that are "touched" is integer and so is the solution.

How to check if there is a point which is not inside any rectangle? Point must be inside of the starting rectangle (0, 0, width, height) and rectangles are inside of the starting rectangle so we are going to find area of union of rectangles and check if it equals to area of the starting rectangle (height * width). If it differs then there exists some point A which is not in any of the rectangles, but also not lying on the edges of rectangles, which should be allowed. This issue can be solved using the fact that all numbers are integers thus if we expand rectangles by value $k-1$ and there is a point $A(x, y)$ which is not inside any rectangle and not lying on the edges of any rectangle then there is a point $B(\text{ceil}(x), \text{ceil}(y))$ which in the worst case would lie on one of the sides of some rectangle and be the top-right corner of square S with side length equal to k .

Since we know how to check if solution with value k is possible, we could use binary search to iterate through solutions and check whether or not the solution is possible, and output the highest possible solution.

Union of the rectangles:

Given N axis-aligned rectangles, calculate the area of their union. We can solve it using sweep line algorithm where events are left and right edges of the rectangles. When we cross the left edge, the rectangle is added to the active set. When we cross the right edge, it is removed from the active set. We now know which rectangles are cut by the sweep line, but we actually need to know what is the length L of the sweep line parts which are intersected by these active rectangles. After processing the event, we can add distance between last two sweep line events multiplied by L to union area.

Every time we add rectangle to the active set, we actually insert vertical segment $[A, B]$ which is representing the rectangle, in some data structure. Length of the sweep line parts which are intersected by active rectangles is length of the union of segments in the data structure.

We need a structure with three main methods `addSegment(A, B)`, `removeSegment(A, B)` and `getUnionOfSegments()`. It can be solved using segment tree data structure. Let's create base segments first, intervals such that no horizontal edge crosses them, except in endpoints, as shown in the image 2.

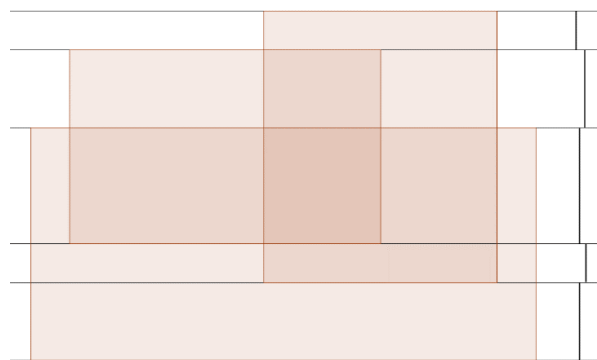


Image 2. Segments on the right are representing base segments

Each leaf node of the segment tree will represent one base segment. Every other node will represent segment of the left child merged with segment of the right child, $[A, B] = \text{merge}([A, C], [C, B])$. Also, for each node we

will save two more variables, length of the union of all segments in the data tree which intersect with segment $[A, B]$ and number of whole segments $[A, B]$ in the segment tree at the moment.

Updating when segment is added in the tree is done using the following algorithm:

1. We are starting from root node and inserting segment $[A, B]$
2. If segment $[C, D]$ which is covered by current node equals to $[A, B]$ then whole segment $[C, D]$ is covered for use so length of union of all segments under the current node is equal to $[A, B]$ and number of whole segments $[A, B]$ is increased by one
3. Otherwise, find intersection $[C, D]$ of segment $[A, B]$ with segment of left child. Call the function recursively for left child and segment $[C, D]$. Do the same for right child.
 - 3.1. Update length for the current node so that it equals to sum of length for both children.

Updating when segment is removed from the tree is done using the similar algorithm, except here length is updated after number of whole segments $[A, B]$ is equal to zero after removing. If node is leaf then it is zero, otherwise it is the sum of lengths for both children.

We should notice that number of nodes updated on each level will be at most four due to the fact that parent node represents merged segments for children nodes. When updating root node, and segment we are inserting is intersecting both left and right parts of the tree, we will visit both children nodes. If not then we will visit appropriate child node and do the same. After that for the left part of the tree, if there exists an intersection with right child then it will be segment $[C, D]$ which is whole segment covered by the right child thus no nodes in right child sub-tree will be visited. It is analogous for the right part of the tree.

Time complexity of the update methods is $O(\log N)$ with constant of four. We are iterating through each of $2N$ events for sweep line and calling update method for each event which gives us total complexity, for checking whether or not the solution k exists, of $O(N \log N)$.

Time complexity: $O(N \log N \log \min(\text{width}, \text{height}))$, where $\log \min(\text{width}, \text{height})$ is binary search complexity for iterating through solutions.

Memory complexity: $O(N)$.

Problem F: Scammer

Authors

Vanja Petrović Tanković

Implementation and analysis

Demjan Grubić

Vanja Petrović Tanković

Bub Bubbles is a scammer and criminal who specializes in money counterfeiting. He lives in Bubbleland. Bubbleland's currency is bubblar, famous for being impossible to counterfeit. Bub was aware of that, so he had to come up with a different technique to get bubblars. He made a machine that can copy any other currency perfectly and print large quantities of it. He could then go to the bank with the fake money and convert it to bubblars.

However, due to economic crisis, banks in Bubbleland exchange currencies in a non-standard way. The exchange rate can be both positive and negative, and they give a fixed bonus amount of money regardless of the amount of money being changed. Different currencies usually have different exchange rates, but not necessarily. Fixed bonuses are, however, always different. For the benefit of their customers who are not good with mathematics, banks do not allow exchanges where the amount of money bank has to pay to the customer is zero or negative (it would mean that the customer would only lose money). Also, the bank has a limited amount of bubblars. Customer cannot exchange amount of money in foreign currency that would result in amount of bubblars exceeding the bank limit.

Bub's machine is powerful. Bub can just enter some positive real number which represents the amount of money machine should print and the machine would calculate what would be the best currency to print the money in. The best currency is, of course, the one that would get him the highest amount of bubblars (not higher than the bank limit). But, for some amounts of money, there might exist multiple best currencies. Bub forgot about that when he made the machine, so when he enters a number for which there is more than one best currency, the machine breaks down.

Your task is to calculate how many different amounts of money would break the machine down.

Input

The first line contains one integer N and one real number M – number of different currencies and the bank limit. Each of the next N lines contains two real numbers r_i and b_i , representing the exchange rate and fixed bonus for the i^{th} currency. All the real numbers in the input contain exactly two fractional digits.

Output

Output contains one integer representing the number of different amounts of money for which there exists more than one currency that Bub could convert in the bank to get the highest amount of bubblars.

Constraints

- $1 \leq N \leq 10^5$
- $5 \leq M \leq 10^6$
- The largest amount of money machine can print is 10^5
- $-10^3 \leq r_i \leq 10^3$
- $0 \leq b_i \leq 10^6$
- No test case will have more than two best currencies for any amount of money

Example input

```
4 6.00
2.00 3.00
-1.00 9.00
0.00 1.40
4.50 0.50
```

Example output

```
2
```

Example explanation

Printing and converting 1 in the first and fourth currencies gives the same highest amount of bubblars ($2.00 \cdot 1 + 3.00 = 4.50 \cdot 1 + 0.50 = 5$). Also, printing and converting 7.6 in the second and third currencies gives the same highest amount of bubblars ($-1.00 \cdot 7.6 + 9.00 = 0.00 \cdot 7.6 + 1.40 = 1.40$). Note that while printing and converting 0.2 in the third and the fourth currencies gives the same amount of bubblars, the first currency gives a higher amount of bubblars not higher than the bank limit.

Solution and analysis:

This task is a geometry problem. It is not hard to see that the currencies are actually straight lines. Money exchange is a linear equation $y = kx + b$, where k is the exchange rate, b is the fixed bonus, x is amount of money given to bank and y is amount of bubblars. Translated to geometry, k is the slope of the line and b is the y -intercept. The bank limit is a horizontal line of the form $y = M$. The best currency for some amount of money x is the line that has the largest y such that it is less than or equal to M . Obviously, we are asked to count certain intersection points of lines. The intersection point (x, y) that should be counted is the one that satisfies $y > 0$, $y \leq M$, $x > 0$, $x \leq 10^5$ and the line segment linking the intersection point and the point (x, M) does not intersect any other lines.

Brute force solution is straightforward – find the intersection point of each pair of lines and test if it satisfies the necessary conditions. There are $O(N^2)$ intersection points and testing if there is a third line that intersects the corresponding line segment has complexity $O(N)$ in the worst case, so the total complexity is $O(N^3)$, which is not nearly good enough.

The appropriate solution to this problem uses divide and conquer technique. To explain the algorithm, we first define what is a *chain*. *Chain* for a set of lines is a set of points such that a point (x, y) lying on any of the lines in the set is in the *chain* if and only if $y > 0$, $y \leq M$, $x > 0$, $x \leq 10^5$ and the line segment linking the point (x, y) and the point (x, M) does not intersect any line from the set (except at the point (x, y)). *Chain* for some set of lines is shown in the Figure 1.

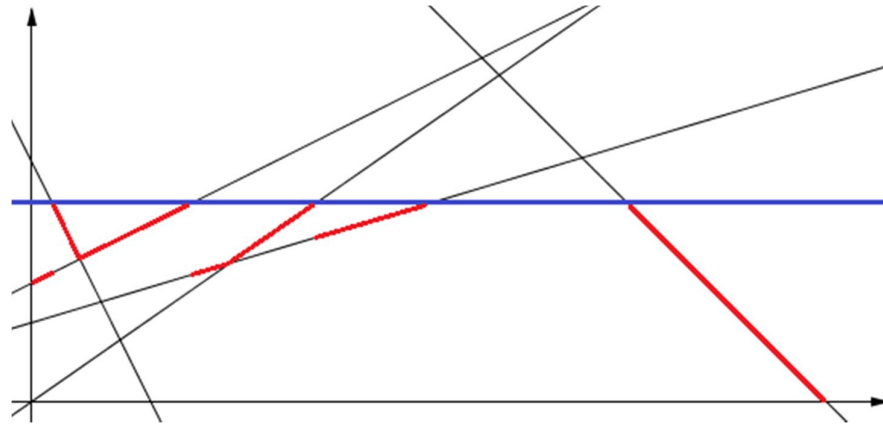


Figure 2. Bank limit is shown in blue and *chain* is shown in red color

Obviously, it is a set of line segments that can be sorted by the x coordinate of the start point (or end point) in a unique way. Number of intersections of two line segments, which can happen at the endpoint of one and start point of the other segment, in *chain* is the solution to the problem for that specific set of lines.

If we are given two *chains*, we can merge them and get a new *chain*. Let's call this operation *mergeChains*. To merge two chains, assume that the start and end points of line segments in each chain are sorted with respect to x coordinate. Just like in *mergesort*, we are going to keep two pointers, pointing to the current point in each chain. The pointers move parallelly – at each step, we are going to increment the pointer that points to the point with lower x coordinate. By keeping track of which *chain* is currently higher (larger y coordinate) and carefully processing intersections of line segments of these two chains, we can build a new *chain*. Notice that the start and end points of this new *chain* are already sorted after this operation is finished. The complexity of *mergeChains* operation is $O(K)$, where K is number of points in the *chain*. An important fact (which you can prove as an exercise) is that $K = O(N)$, where N is number of lines forming a *chain*. Therefore, the complexity of this operation is $O(N)$. This also means that the solution for the whole problem, the number of relevant intersections, is $O(N)$.

Now, the goal of our algorithm is to build the *chain* for the whole set of lines. At the beginning, each line forms a *chain* containing at most one line segment. Therefore, we start with N chains. Using *mergeChains* to merge first and second *chain*, then third and fourth, then fifth and sixth and so on, we end up with $\lceil N/2 \rceil$ chains. Repeating this process, after $O(\log N)$ steps we end up with one final chain which we can use to count the total number of relevant intersections. Each step has the complexity $O(N)$, so the total complexity of this algorithm is $O(N \log N)$.

Problem G: Unary

Authors

Vanja Petrović Tanković

Implementation and analysis

Aleksandar Milovanović

Vanja Petrović Tanković

Unary numeral system (base-1) is a numeral system where a number K is represented with an arbitrarily chosen symbol repeated K times. The chosen symbol for this task is digit 1. Digit 0 will be used as a separator between two numbers.

Writing positive integers in unary system consecutively (using 0 to separate them), you get a sequence of digits

101101110111101111101111110 ...

By removing every second digit 1 from this sequence, you obtain a new sequence:

10~~1~~10~~1~~1~~1~~01~~1~~1~~1~~01~~1~~1~~1~~101~~1~~1~~1~~101~~1~~1~~1~~10 ... \rightarrow 10101011011101110 ...

Given N , determine the N^{th} digit of the resulting sequence.

Input

The first and only line of input contains one positive integer N ($1 \leq N \leq 10^{18}$).

Output

Output contains one character, either '0' or '1', representing the N^{th} digit of the sequence.

Example input

6

Example output

0

Solution and analysis:

First, let's see how do we determine the N^{th} digit of the sequence without removing every second digit 1. After writing exactly K numbers in the unary system consecutively, we are going to have exactly K zeros as separators, while the number of ones is going to be

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

It follows that the K^{th} zero in this sequence is at the position $K + \frac{K(K+1)}{2}$. Therefore, if the positive solution for K in the following quadratic equation is integer, the N^{th} digit is 0, otherwise it is 1.

$$K + \frac{K(K+1)}{2} = N$$

When we remove every second digit 1, the number of ones decreases by

$$\left\lfloor \frac{\frac{K(K+1)}{2}}{2} \right\rfloor.$$

Including the previous term in the quadratic equation, we get

$$K + \frac{K(K+1)}{2} - \left\lfloor \frac{K(K+1)}{4} \right\rfloor = N.$$

This equation can also be solved for K to check if the N^{th} digit is either 0 or 1, but it is a bit trickier. Also, the standard form for solving quadratic equation uses *sqrt* function. This is an implementation problem because we get a term containing N in the *sqrt* and N can be as large as 10^{18} . The precision of *double* data type is not good enough to perfectly represent such large numbers.

One way of solving this is to use the solution to quadratic equation as an approximation, and then to test a few integers smaller and a few integers larger than the approximated solution if they fit in the equation. If one of those integers is a solution, the N^{th} digit is 0, otherwise it is 1. If we ignore the complexity of the *sqrt* function, this solution is $O(1)$.

Another solution that is safer than the previous and uses only integers is to use binary search. By using binary search on K , we can find the largest K such that

$$K + \frac{K(K+1)}{2} - \left\lfloor \frac{K(K+1)}{4} \right\rfloor \leq N.$$

If such K satisfies the equation $K + \frac{K(K+1)}{2} - \left\lfloor \frac{K(K+1)}{4} \right\rfloor = N$, the N^{th} digit is 0, otherwise it is 1. The complexity of binary search is logarithmic, which is a bit more complex than the previous solution. However, this method is stable and does not deal with floating-point arithmetics. Note that it is important to set the upper limit of the search to a number that is not much larger than $2 \cdot 10^9$, so that it is large enough to find the solution, but not too large to cause overflow when using the 64-bit integer data types.

Problem H: Graffiti

Authors

Demjan Grubić

Implementation and analysis

Demjan Grubić

Nikola Stojiljković

Young artist Petr likes to draw graffiti. He just have found a long wall which consist of N consecutive empty places, numbered from 1 to N , where he can draw graffiti, and he wants to fill the whole wall with his masterpieces. Every hour he wants to draw a new graffiti on another place on the wall, and he has already decided in which order he is going to do that. When he draws a graffiti at place i on the wall, and he wants to draw next graffiti at place j , he walks from place i to the place j and he passes by all places between i and j .

Interesting about young Petr is that he loves his work, and every time he passes by his graffiti, he has to take picture of it. He doesn't take a picture of graffiti that he has just drawn.

As said, he has already decided in what order he is going to draw graffiti and now he asks you to help him to find out how many times he is going to take picture of each graffiti.

Input

The first line of standard input contains number N ($1 \leq N \leq 100\,000$), number of places on the wall. Next line contains N distinct numbers from interval $[1..N]$ where i -th number represent the place on the wall where young Petr is going to draw graffiti at i -th hour.

Output

On the first and only line of standard output you should print N numbers where i -th number represent the number of times Petr is going to take a picture of the graffiti that is going to be drawn at i -th place on the wall.

Example input

```
5
2 4 1 5 3
```

Example output

```
0 2 0 2 0
```

Solution and analysis:

If Petr walks from position i to position j , we are going to consider interval $[i, j]$. For given input we are getting $N - 1$ intervals. Now for each position i we have to find out how many intervals contains number i , but we take in consideration only intervals after the one that starts with number i .

If there weren't this last condition, if we were only asked to calculate for every position i how many intervals contain that number, we could solve it using well known data structure Segment Tree where each node would represent how many times we crossed that whole interval, but we haven't crossed whole interval represented by its father's node. After updating segment tree with each interval, the result for number i

would be sum of values in nodes on the path in segment tree from root to the leaf that represents number i , let's call that sum $SegmentTreeQuery(i)$. This is known as lazy propagation in Segment Tree.

After updating segment tree with i -th interval (let's say that interval is $[l, r]$), we should find $SegmentTreeQuery(r)$ and remember that in array $before[r]$. After inserting all intervals in segment tree, results for i -th position is $SegmentTreeQuery(i) - before[i]$.

Qualifications

The qualifications were split into two rounds, with ten problems in each round. This year new type of problem is introduced - challenge problems, which are problems that do not have a known "best" solution. Outputs for these problems compete against each other, and scores are scaled according to the best solution from the contestants. In the first round, maximal number of points one team could get on challenge problem was 4, while in the second round each challenge problem was worth 8 points. Like in previous years, every non-challenge problem in the first round was worth 1 points, and each non-challenge problem in the second round was worth 2 points.

The problems for both rounds were chosen from the publicly available archives at the Spoj site (www.spoj.pl).

This year, 86 teams managed to solve at least one problem from the qualifying rounds. We are especially proud of the fact that not only we have participants from the region (Bulgaria, Croatia, Montenegro, Serbia) but also teams from Germany, Lithuania and Poland fought their way to the Finals.

Num	Problem name	ID	Accepted solutions
01	A Famous Grid	11582	78
02	Projections Of A Polygon	1431	39
03	Circular game	4309	19
04	Two Famous Companies	11579	36
05	Special Graph	13529	36
06	Dinosaur Menace	7187	48
07	The Ball	8391	53
08	Palindromes	9748	49
09	Shrinking Polygons	3415	71
10	[CH] Tetris AI	758	41

Table 1. Statistics for Round 1

Num	Problem name	ID	Accepted solutions
01	Avoiding SOS Grids	6999	21
02	High and Low	12210	25
03	Cut on a tree	13369	Not solved
04	Tower Game (Hard)	7857	28
05	Fibonacci recursive sequences (hard)	12009	6
06	Help Blue Mary Please! (Act I)	1457	6
07	Problems Collection (Volume X)	1815	34
08	AB-words	177	9
09	[CH] Santa Claus and the Presents	240	37
10	[CH] Robo Eye	2629	21

Table 2. Statistics for Round 2

This year we wanted to continue with our tradition that the contestants are the ones who are writing the solutions for qualifications problems. You should note that these solutions are not official - we cannot guarantee that all of them are accurate in general. (Still, a correct implementation should pass all of the test cases on the Spoj site.)

The organizers would like to express their gratitude to everyone who participated in writing the solutions.

Problem R1 01: A Famous Grid (ID: 11582)

Time Limit: 3.0 second

Memory Limit: 256 MB

Mr. B has recently discovered the grid named "spiral grid". Construct the grid like the following figure. (The grid is actually infinite. The figure is only a small part of it.)

100	99	98	97	96	95	94	93	92	91
65	64	63	62	61	60	59	58	57	90
66	37	36	35	34	33	32	31	56	89
67	38	17	16	15	14	13	30	55	88
68	39	18	5	4	3	12	29	54	87
69	40	19	6	1	2	11	28	53	86
70	41	20	7	8	9	10	27	52	85
71	42	21	22	23	24	25	26	51	84
72	43	44	45	46	47	48	49	50	83
73	74	75	76	77	78	79	80	81	82

Considering traveling around it, you are free to any cell containing a composite number or 1, but traveling to any cell containing a prime number is disallowed. You can travel up, down, left or right, but not diagonally. Write a program to find the length of the shortest path between pairs of nonprime numbers, or report it's impossible.

			97						
				61		59			
	37						31		89
67		17				13			
			5		3		29		
		19			2	11		53	
	41		7						
71				23					
	43				47				83
73						79			

Input

Each test case is described by a line of input containing two nonprime integer $1 \leq x, y \leq 10,000$.

Output

For each test case display its case number followed by the length of the shortest path or impossible in one line.

Sample

input	output
1 4	Case 1: 1
9 32	Case 2: 7
10 12	Case 3: impossible

Solution:

This task was by far the easiest one in this year's edition of BubbleCup.

Since the input numbers are small, $1 \leq x, y \leq 10,000$, we know that the maximum possible number in input will be located within the grid of size 100, so we can construct the grid of size 128 to allow the shortest path that we are looking for to go outside the smaller grid. Construction of the grid is very simple, just fill the first

field with number $128 \cdot 128 = 16384$ and then fill the remaining numbers in clockwise order. After that we must mark all fields which contain prime numbers as impassable. We can use the **sieve of Eratosthenes** to filter out the prime numbers. Once we have constructed the grid, we can use **BFS (breadth-first search)** to find the answer for each test case.

The time complexity of the solution is **linear** compared to the sum of distances between pairs of fields given in the input.

Solution by:

Name: Aleksandar Ivanović

School: University of Belgrade, School of Electrical Engineering

E-mail: aleksandar.ivanovic.94@gmail.com

Problem R1 02: Projections Of A Polygon (ID: 1431)

Time Limit: 1.0 second

Memory Limit: 256 MB

You are given a convex polygon on Cartesian coordinate system. It has projections on X and Y -axis. You can arbitrary rotate this polygon. What minimum and maximum sum of projections can you achieve?

Input

First line contains one integer number N ($3 \leq N \leq 100$) - number of polygon's vertices. Following N lines contain vertex coordinates X_i and Y_i . All numbers are integers. Vertices are given in clockwise or anticlockwise direction. No two vertices coincide. No three consecutive vertices lie on the same line. All coordinates do not exceed 10000 by absolute value.

Output

Write minimum and maximum value of sum of the polygon's projections. Separate them by a space. Your answer should not differ with the correct one more than 10^{-6} .

Sample

input	output
4 0 0 0 1 1 1 1 0	2 2.828427124

Solution:

The First thing we can conclude is that every rotation of $k \cdot \pi/2$ angles would give the same result. So, the solution angles are somewhere between 0 and $\pi/2$.

Other thing we can notice is that for a given angle of rotation, we can describe a polygon with 4 points – the left one, the top one, the right one and the bottom one – the result is $top.y - bottom.y + right.x - left.x$. This means that we can divide our search space $(0, \pi/2)$ into smaller sections in which the left, bottom, up and right point don't change (when any of them change we call it a new section). There are at most $4n$ of these sections (even less, if we watch only the first quadrant) because, since the polygon is convex, it's impossible for a point to be on one extreme more than once.

The nature of result for each of the individual sections is such that the solution rises until one point and then drops (or falls until one point and then rises). Because of that fact, and the fact that we can easily check for a value at a given angle, we can search through these sections with **ternary search** and find the min and the max solution for a section. Final solution is the max (min) from all the section results.

Therefore our algorithm works as follows:

- Starting from the polygon given in the input, we find four points that define the rounding box for this polygon, and after that we can easily calculate the angle after which at least one of these points change (some other 4 points define the rounding box). By doing this repeatedly, we can find all the sections in $O(n)$, because transition between sections can be done in $O(1)$ and there are $O(n)$ sections. The transition is done in constant time by noting that, since polygon is convex, only points adjacent to 4 points that define rounding box are candidates for next 4 points that define rounding box.
- Once we find the sections, we perform a ternary search in order to find the best places for min and max solution, and we can do this in $O(\log n)$ time complexity.

Therefore, the overall time complexity is $O(n \log n)$. Implementing this idea is tricky, and since $n \leq 100$, it is recommended to slow down some part of algorithm in order to make the implementation of that part easier.

Solution by:

Name: Aleksandar Milovanović

School: Faculty of Computing

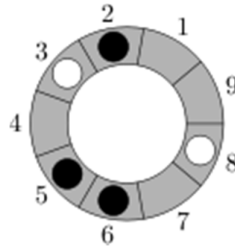
E-mail: zaxa321@gmail.com

Problem R1 03: Circular game (ID: 4309)

Time Limit: 2.0 second

Memory Limit: 256 MB

In the "circular game" the board consists of m fields arranged on a circle and numbered from 1 to m . On the board there are b white and c black pieces, at most one on each field. Two players are playing the game, the white player and the black player. Starting from the white player, the players perform their moves on the board alternately. A move consists of moving a piece of the player's colour any number of free fields forward or backward. For instance, for the board depicted below, the white player can move the piece from field 3 to field 4 or the piece from field 8 to any of the fields 7, 9 and 1.



If a player can perform no moves in his turn, he loses. Knowing that both players play optimally, check who wins the game. It can happen that none of the players wins (the game never ends).

Input

The first line of the standard input contains one integer t representing the number of boards to be considered.

The following lines contain descriptions of respective boards, each of which consists of three lines. In the first line there are three integers m , b and c ($1 \leq m \leq 109$, $1 \leq b, c$) separated by single spaces and denoting the length of the board, the number of white pieces and the number of black pieces. In the second line there is an increasing sequence of b integers (in the range $1, \dots, m$) representing the positions of white pieces. In the third line there is an increasing sequence of c integers (in the range $1, \dots, m$) representing the positions of black pieces. The total number of pieces in all boards does not exceed 10^6 .

Output

Exactly t lines with answers for consecutive boards should be written to the standard output. The answer is always a single character: B, C, or R, depending on whether the white player wins (B), the black player wins (C) or the game never ends (R).

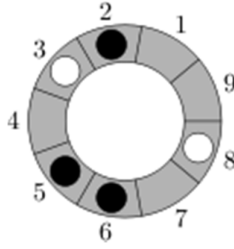
Sample

input	output
3	C
9 2 3	B
3 8	R
2 5 6	
6 2 2	
5 6	
2 4	
7 1 1	
3	
4	

Solution:

This game is similar to the popular game "Nim" (<http://en.wikipedia.org/wiki/Nim>), so in order to understand the solution we recommend you to read about nim-sums.

Let's first define some terms. We will call "block" a sequence of fields on the board, containing pieces of single color, such that first and last field in the sequence contains pieces on them. For every block we will define next attributes: P_i – the number of pieces in the block, N_i – the length of the block, L_i – the number of fields between that block and the next block (in counter wise direction). So for example, in the example picture we have next three blocks(starting from field 2): (1,1,0), (1,1,1), (2,2,1), (1,1,2).



It is easy to see that if there are no free slots on the board - the black wins. In addition, if all blocks have $P_i = 1$ - then it is a tie, and when only one player has all blocks where $P_i = 1$ - he loses. Next we assume that each player has at least one block of P_i greater than 1.

Let's assume first that all blocks have $P_i = 2$. The nim-sum is now:

$$s = L_1 \oplus L_2 \oplus \dots \oplus L_n$$

Similar as in the Nim game(taking k objects from the heaps in the Nim game \Leftrightarrow moving last piece in the block k fields counter wise), if $s = 0$ then black wins, otherwise white wins.

The things get complicated when some blocks have P_i different than 2. Let's define new term, the strong block is the block which has $3 \leq P_i < N_i$. Notice that player with the strong block always has the possibility of moving some piece inside block, so he cannot lose. So, if both players have at least one strong block, it is a tie. So, if player wants to win, he can't allow the enemy to form the strong block (if he doesn't have it already).

So, in the beginning, white player plays first, and let's define S_w and S_b numbers of white and black blocks which can become a strong blocks in a single move. Now we have next three cases:

1. Black has a strong block, or $S_b > 1$, so the white player is not able to prevent black from forming the strong block. If white player has a strong block, or $S_w > 0$, than it is a tie, otherwise black wins.
2. $S_b = 0$, then if white has a strong block, or $S_w > 0$ white wins. Otherwise, win depends on the nim-sum s (if $s = 0$ then black wins, otherwise white wins).
3. $S_b = 1$, so to win white player must prevent black player from forming the block, or create the strong block himself to get the tie. Subcases:
 - 3.1. White player can prevent black player from forming the block. Now we have next cases:
 - 3.1.1. $S_w > 1$, white player wins.
 - 3.1.2. $S_w = 1$, if black player can prevent white from forming the strong block(in the second move, after white player spends his first move to prevent black player from forming the strong block) then we check who wins depending on the nim sum s (if $s = 0$ then black wins, otherwise white wins), else white wins.
 - 3.1.3. $S_w = 0$, we check who wins depending on the nim sum s (if $s = 0$ then black wins, otherwise white wins), else white wins.
 - 3.2. White player can't prevent the black player from forming the strong block. Now we have next cases:
 - 3.2.1. $S_w > 0$, white player forms the strong block too, so it is a tie.
 - 3.2.2. $S_w = 0$, black player wins.

Solution by:

Name: **Marko Rakita**

School: PMF

E-mail: marko21rakita@gmail.com

Problem R1 04: Two Famous Companies (ID: 11579)

Time Limit: 53.0 second

Memory Limit: 256 MB

In China, there are two companies which offer the internet service for the people from all cities: China Telecom and China Unicom. They both are planning to build cables between cities. The government wants to connect all the cities in minimum costs of course. So the minister of finance Mr. B wants to choose some of the cable plans from the two companies and calculate the minimum cost needed to connect all the cities. Mr. B knows that there are $N-1$ cables should be built in order to connect all N cities of China. For some political reason, Mr. B should choose K cables from the China Telecom and the rest $N-1-K$ cables from the China Unicom. Your job is to help Mr. B determine which cables should be built and the minimum cost to build them. You may assume that the solution always exists.

Input

Each test case starts with a line containing the number of cities N ($1 \leq N \leq 50,000$), number of cable plans M ($N-1 \leq M \leq 100,000$) and the number of required cables from China Telecom K ($0 \leq K \leq N-1$). This is followed by M lines, each containing four integers a, b, c, x ($0 \leq a, b \leq N-1, a \neq b, 1 \leq c \leq 100, x \in [0,1]$) indicating the pair of cities this cable will connect, the cost to build this cable and the company this cable plan belongs to. $x = 0$ denotes this cable plan belongs to China Telecom and $x = 1$ denotes this cable plan is from China Unicom.

Output

For each test case, display the case number and the minimum cost of the cable building plan.

Sample

input	output
2 2 1 0 1 1 1 0 1 2 0 2 2 0 0 1 1 1 0 1 2 0	Case 1: 2 Case 2: 1

Explanation

In the first case, there are two cable plans between the only two cities, one from China Telecom and one from China Unicom. Mr. B needs to choose the one from China Telecom to satisfy the problem requirement even the cost is higher.

In the second case, Mr. B must choose the cable from China Unicom, which leads the answer to 1.

Solution:

This problem is very similar to finding the minimum spanning tree (MST). It is a well-known problem, where you are given a weighted graph and you are asked to pick edges with minimal sum such that those edges span the whole graph (there is a path between each pair of vertices).

One way to find MST is to sort all edges by weight and to take them one by one. If the connected vertices are in the same structure (which is itself a tree, of course), we discard the edge, otherwise, we use it and join the vertices. We end up with a single tree.

It is a well-known fact that this greedy solution works. This algorithm is named after Kruskal.

To maintain connected components we use Disjoint-set data structure. It is easy to code it. We maintain two

arrays: parents of vertices in the tree (with itself as a parent if the vertex has no parent) and sizes of the trees (to avoid tall trees) (let $p[]$ represent the former and $size[]$ the latter array).

Two components are joined in this way:

```

=====
Function: getParent
Input:  v - index of a vertex
Output: parent of a given vertex
-----

01  if p[v] == v then
02      return v
03  else
04      p[v] = getParent(p[v])
05      return p[v]
-----

Function: unionSet
Input:  a - index of the first vertex
       b - index of the second vertex
       (vertices a and b are in two different components)
-----

06  parent_a = getParent(a)
07  parent_b = getParent(b)
08  If size[a] >= size[b] then
09      p[parent_b] = parent_a
10      size[parent_a] += size[parent_b]
11  Else
12      p[parent_a] = parent_b
13      size[parent_b] += size[parent_a]
=====

Pseudo code for finding Disjoint-set data structure

```

As all prerequisites are met we can focus on our problem. If we ignore the number of the company and we run Kruskal's algorithm we probably end up with building more cables from one company than required. It is important to notice that the cost is optimal but we do not need exactly that.

Let's look at the cables from one company (for example, Telecom).

We take more cables than needed because they are cheaper. There is an elementary (although difficult to come up with) trick here – we can increase the cost of cables, which will force us to take less of them. The bigger the cost, the less cables we take. Relative cost between the cables must remain the same (for example, equal cables should have the same priority). So, let's add a cost *dif* to each cost of Telecom built cables.

If *dif* is very small we take all the cables we could take, if it is very big, we discard all cables we could discard. Notice that we will find the value of *dif* such that we take no more than k (given in the statement) cables. I say "no more" because we might take not exactly k Telecom cables when adjusting *dif* value.

It turns out that it is very easy to brute force all *dif* values (adding -101 makes the best Telecom cable worse than the worst Unicom cable and adding 101 makes the exact opposite). This is due to the fact that given costs are integers in the interval [1; 100].

So let's check all *dif* values and run Kruskal with that (in addition, we need to check how many Telecom cables we take).

The most important function in the solution with modified Kruskal's algorithm:

```

=====
Function: getCost
Input:  dif - cost that will be added to Telecom cables
Output: res - cost of MST
       taken - number of Telecom cables taken
-----

01  res = taken = 0
02  i = 0; j = 0;
03  while (i < tlen) and (j < ulen) do

```

```

04         if (T[i].cost + dif <= U[j].cost) then
05             a = T[i].first_vertex
06             b = T[i].second_vertex
07             c = T[i].cost
08             If (getParent(a) <> getParent(b)) then
09                 unionSet(a, b)
10                 res += c
11                 taken += 1
12             i += 1
13         Else
14             a = U[j].first_vertex
15             b = U[j].second_vertex
16             c = U[j].cost
17             If (getParent(a) <> getParent(b)) then
18                 unionSet(a, b)
19                 res += c
20             j += 1
21     While (i < tlen) do
22         a = T[i].first_vertex
23         b = T[i].second_vertex
24         c = T[i].cost
25         If (getParent(a) <> getParent(b)) then
26             unionSet(a, b)
27             res += c
28             taken += 1
29         i += 1
30     While (j < ulen) do
31         a = U[j].first_vertex
32         b = U[j].second_vertex
33         c = U[j].cost
34         If (getParent(a) <> getParent(b)) then
35             unionSet(a, b)
36             res += c
37         j += 1

```

$(T, tlen), (U, ulen)$ are sorted arrays of Telecom and Unicom cables, respectively. We just merge them comparing modified weights and accumulating the real minimum spanning tree cost res .

We need only to use the function properly:

```

=====
01     For i = -MaxD to MaxD do
02         getCost(i, res, taken)
03         If taken <= k then
04             Print "Case " + case + " " + (res - (k - taken) * (i-1))
05             break
=====

```

$Maxd$ is a constant equal to a bit more than the maximum edge cost (105).

The last thing that needs an explanation is the expression: $(k - taken) * (i - 1)$. Its meaning is that we take $(k - taken)$ more cables of Telecom with $dif = i - 1$ (so they are $(i - 1)$ units smaller then the corresponding Unicom cables).

Solution by:

Name: *Karolis Kusas*

School: *Kaunas University of Technology*

E-mail: *karolis.kusas@gmail.com*

Problem R1 05: Special Graph (ID: 13529)

Time Limit: 0.5-1.0 second

Memory Limit: 1536 MB

You are given a directed graph with N vertices. The special thing about the graph is that each vertex has at most one outgoing edge. Your task is to answer the following two types of queries

1 a - delete the only edge outgoing from vertex a . It is guaranteed that the edge exists. $1 \leq a \leq N$

2 $a b$ - output the length of the shortest path from vertex a to vertex b , if the path exists. Otherwise output "-1" without quotes. $1 \leq a, b \leq N$.

Input

First line of input contains a natural number $N \leq 10^5$ the number of vertices in the graph.

The following line contains N integer numbers, i -th number is $[i]$ ($0 \leq next[i] \leq N$), meaning that there is an edge from vertex i to vertex $next[i]$. If $next[i] = 0$, assume that there is no outgoing edge from vertex i .

Third line contains a natural number $M \leq 10^5$ the number of queries.

The following M lines contain a query each. Queries are given in the manner described above.

Output

On the i -th line output the answer for the i -th query of type 2 $a b$.

Sample

input	output
6	4
3 3 4 5 6 4	2
6	-1
2 1 6	-1
2 1 4	-1
2 1 2	
1 3	
2 1 6	
2 1 4	

Solution:

This is a rather interesting problem that can cause quite a headache in the implementation stage. For starters, note that the graph we are given here represents a **partial function** of a set of vertices V to itself (each member maps to at most one other). Let $p: V \rightarrow V$ be this function. We can make it a **total function** (each member maps to exactly one other) by adding a node '0' to the set (let V_0 be this new set) and a few extra edges. Let's call this function $f: V_0 \rightarrow V_0$ and define it as follows:

$$f(x) = p(x), x \in V \wedge \exists p(x)$$

$$f(x) = 0, \text{ otherwise}$$

It would be useful if we could efficiently calculate $f^k(x)$, which represents the k -th node on the path starting with x . To do this, we can note that composition of a function to itself behaves additively: $f^a(f^b(x)) = f^{a+b}(x)$. What we need to do is pre-compute $f^k(x)$ for $k = 1, 2, 4, 8 \dots$ and other degrees of two. This can be easily done in $O(n \log n)$ time complexity using the fact that $f^{2^a}(x) = f^{2^{a-1}}(f^{2^{a-1}})$. After this initial pre-computation, we can calculate $f^k(x)$ for any k in $O(\log n)$ time by observing the binary representation of k .

Therefore, with this in mind, we can easily find in $O(\log n)$ time whether distance between node a and node b is x , by checking whether $f^x(a) = b$.

Since every node can have at most one outgoing edge, each connected component of this graph is either a tree or a cycle with trees connected to it.

Let's, for a moment, rule out the possibility of cycles and edge deletions. How can we effectively determine the (shortest) path length between two nodes A and B if it exists? (note that it is allowed to give any answer to the query if there is no path, because we can use the work mentioned above to quickly verify our answer). One way to do this is by labeling each node in the graph. To do this, we perform a **depth-first search** of the graph. Every time we start exploring a new path, we label the first node with 0, the next one with 1, and so on. This goes on until we hit a node without outgoing edges, or until we hit an already explored node. In the latter case, let x be the label on the node we have hit; what we do then is re-label the nodes in the current path in reverse: the last unexplored node gets labeled with $x - 1$, the next one with $x - 2$, and so on. It is now clear that, if a path exists between two nodes, its length is equal to the difference between their labels. This, in combination with the method of quickly verifying the correctness of this answer, gives us an efficient algorithm for the acyclic case without edge deletion.

To handle edge deletion, we will use the offline approach. After reading all the queries, we will answer them in reverse order. By doing that we have turned edge deletion to edge addition, and with Union-Find structure we can easily see whether two nodes are in the same connected component, and therefore can easily see whether there is an edge deleted on a path between two nodes. Beside this, for each cycle, we should maintain some kind of self-balancing binary search tree in order to check if there is a deleted edge on a path between two nodes on a cycle.

Finally, in order to implement the possibility of cycles, we should store a relative position of each node in a cycle (we can label starting from any node) in our initial pass. Three possible cases arise in a shortest path query from A to B :

- A and B are both in a cycle: if their cycle is not the same, there is no path; otherwise, use their relative positions to determine the path length;
- A is in a cycle and B is not: since we can never exit a cycle in this kind of graph, the path does not exist;
- A is not in a cycle and B is: Since each connected component of a graph can have at most one cycle, we should first check whether B 's cycle is connected to A 's tree (this information can be pre-computed in our initial pass). If not, there is no path. Otherwise, we can split this path into two parts that we know how to solve. The first path is from A to the entry point to the cycle from A (corresponding to the acyclic case), and the second path is from the entry point to B (corresponding to the case with two nodes in a cycle).

Overall, our algorithm requires $O(n \log n)$ time for pre-computation and relabeling and $O(\log n)$ per query, giving us an overall time complexity of $O((n + m) \log n)$.

Solution by:

Name: **Petar Veličković**

School: *University of Cambridge*

E-mail: *pv273@cam.ac.uk*

Problem R1 06: Dinosaur Menace (ID: 7187)

Time Limit: 9.0 second

Memory Limit: 256 MB

After a failed but interesting DNA project, a lot of dinosaurs spread over the lab devouring most of the staff. Jeff, a scientist that worked in the project, managed to survive by hiding in the southwest corner of the lab. Now that all dinosaurs are asleep, he is going to try to leave. The exit of the lab is located at the northeast corner. Jeff knows that if any of the dinosaurs wakes up, he does not stand a chance, so he needs to minimize the likelihood of that happening. For that, he wants to follow a path that maximizes the minimum distance from him to a dinosaur along the path. The length of the path is of no interest to Jeff. For this problem we consider that Jeff and the dinosaurs are points on the plane, and that Jeff's path is a continuous curve connecting the southwest and northeast corners of the lab. As we mentioned, Jeff wants to maximize the minimum distance between this curve and the position of any dinosaur.

Input

The input contains several test cases, each one described in several lines. The first line of each test case contains three integers N , W , and H separated by single spaces. The value N is the number of dinosaurs in the lab ($1 \leq N \leq 300$). The values W (width) and H (height) are the size of the lab on the x and y coordinates, respectively ($2 \leq W, H \leq 10^6$). This means that the starting position of Jeff is at $(0, 0)$, while the exit of the lab is located at (W, H) . Each of the next N lines contains two integers X and Y separated by a single space, representing the coordinates of a different dinosaur ($1 \leq X \leq W - 1$ and $1 \leq Y \leq H - 1$). Note that no dinosaur is located on the border of the lab. You may assume that no two dinosaurs have the same location. The last line of the input contains the number -1 three times separated by single spaces and should not be processed as a test case.

Output

For each test case output a single line with the maximum possible distance to the closest dinosaur. Write the result rounded to the closest number with exactly three decimal places, using the highest in case of ties, as usual.

Sample

input	output
1 2 2	1.000
1 1	1.581
3 5 4	1.803
1 3	
4 1	
1 2	
2 5 4	
1 3	
4 1	
-1 -1 -1	

Solution:

One of the standard things in problems where we search for minimum or maximum value for which something is possible is to bring it down to an easier version – check if that something is possible for some constant value, and then that value can be found by binary search.

In this task, we will precede this way too: Create a version of the problem where we check if it is possible to get from one side of laboratory to another, keeping the distance from all the dinosaurs to be at least D . And the question is what is the maximum D for which this condition is satisfied.

To solve this easier version, we should rephrase it: When we draw circles with radius D around each dinosaur, can we go through laboratory without going inside of any circle? To test this, we will first transform this problem into a graph theory problem. The vertices of the graph will represent circles. Two vertices are connected if and only if the circles intersect. Then we need to find connected components of this undirected graph. This can be accomplished with a breadth-first search or a depth-first search algorithm.

To check if there is a path from the lower left corner of the room to the upper right corner, we need to test if there exists a component of circles such that it intersects any of the following pairs of walls: top and bottom wall, left and bottom wall, left and right wall or top and right wall. It is obvious that if a component of circles intersects any of these pairs of walls, there exists no path from the lower left corner to the upper right corner of the room.

Recap of the algorithm: Use a binary search to find the maximum distance. For a fixed D in binary search, make circles of radius D centered at dinosaur points. Make a graph of circles, so that two circles are connected if and only if they intersect. Find connected components of the graph. Test if any component intersects certain pairs of walls. Update the binary search interval according to this.

Since there are N dinosaurs, the complexity of graph construction and BFS is $O(N^2)$. Binary search has the complexity of $O(\log(\min(W,H)))$. This brings the total time complexity of the solution to $O(N^2 \log(\min(W,H)))$.

Solution by:

Name: **Vanja Petrović Tanković**

School: Faculty of Computing

E-mail: vpetrovictankovic@gmail.com

Problem R1 07: The Ball (ID: 8391)

Time Limit: 2.0 second

Memory Limit: 256 MB

In a coordinate plane, there are N horizontal conveyor belts, each moving either leftwards or rightwards. When the ball falls on a belt, the belt drags it in direction it's moving. When the ball reaches the end of the belt, it falls vertically downwards. For example, if the belt is moving rightwards and it ends in the unit square with x -coordinate 12, the ball will fall from the belt on the x -coordinate 13, and continue falling on the same x -coordinate until it falls on another belt or reaches the ground (the height of 0).

Frane drops a ball many times (from the height that is greater than any of the belt heights), from various x -coordinates, and your task is: for each ball Frane drops, determine the direction of each belt such that this ball falls on as many belts as possible.

B			B	B	B	
→	→	→	→			
				←	←	←
	→	→	→			

This picture represents the first test example:

Input

In the first line of input, there is an integer N (the number of conveyor belts, $1 \leq N \leq 100\,000$).

In each of the next N lines, there are integers $X1, X2, Y$ ($X1 \leq X2, 0 < X1, X2, Y < 10^9$) representing the belt. Imagine the belt as a segment of which the bounding unit squares are $(X1, Y)$ and $(X2, Y)$. The belt's thickness is zero and it lies on the bottom of the given unit squares. The belts will not touch or overlap each other.

In the next line, there is an integer Q (the number of falling balls, $1 \leq Q \leq 100\,000$).

In the next Q lines there is an integer less than 10^9 , representing the x -coordinate of the unit square Frane drops the ball from.

Output

For each of the Q queries, output the greatest possible number of the conveyor belts visited by the ball.

Sample

input	output
3	3
1 4 3	3
5 7 2	2
2 4 1	2
4	
1	
4	
5	
6	

input	output
-------	--------

3	3
5 20 20	2
15 30 15	0
10 14 11	
3	
5	
30	
516546	

Solution:

In order to solve this task efficiently, we will split it into two parts. In the first part, we will determine the following:

- For each ball, which conveyor belt it hits first
- For each conveyor belt, which belt will the ball fall onto next if the current belt is moving to the left, as well as if it's moving to the right

After this information has been gathered, we will proceed to the second part, which calculates the “score” (i.e. the maximum number of visited belts) for each ball – this is the easier part, therefore we will explain it first.

Since each conveyor belt can move either left or right, there are too many possible combinations, which is why we will use dynamic programming. Namely, we start by sorting the belts in ascending order by their y-coordinate, after which we process them one at a time. For each belt, we observe the next two belts below, which we have previously found in the first part (one for each direction). Since those belts have a lower y-coordinate, they must have already been processed, which means we know the score for any ball dropped onto them. By observing the larger of these two, we can determine whether the current belt should go left or right, and calculate its score by adding 1 to the score of the belt below. Since, eventually, the ball must fall to the ground, we can include an additional, infinite belt lying on the ground with a score of 0. Dynamic programming itself has linear time complexity, but because of the need to sort the belts, the total complexity is $\Theta(n \log n)$. After we have calculated the score for all the belts, we can output the results, since we also know which belt each ball hits at the beginning.

The first part, however, is not that easy. The trivial approach would be to go through all the belts for each ball, as well as for each belt, but the quadratic complexity would lead to TLE – this is why we will use a sweep line algorithm instead. A sweep line algorithm is a geometric algorithm which consists of a vertical line in the plane moving from left to right, but not continuously, since it stops only at certain points which we call events. There can be many different types of events, and different actions can be taken for each one of those.

For this task, we will introduce five types of events and enumerate them in the following order:

1. A point in the plane which represents the start of a belt
2. A point in the plane where a ball starts falling from a belt to the left
3. A point in the plane where a ball starts falling from a belt to the right
4. A point in the plane from which a ball is dropped
5. A point in the plane which represents the end of a belt

Numbers 1 and 5 are obviously the starting and ending points of a belt; 2 and 3 are the points right next to those (i.e. $(x_{start} - 1, y)$ and $(x_{end} + 1, y)$); finally, for number 4 we only know the x-coordinate, so we assign it a large y-coordinate, above all the belts.

We will, at all times, also maintain a list of all the belts the sweep line currently intersects. Ideally, this should be stored in a **self-balancing binary search tree**, BST, sorted by y-coordinate for fast search, insertion and

deletion.

Now that we have all the information, we start by sorting the events by their x-coordinate (because the sweep line moves from left to right). If two events have the same x-coordinate, we sort them by their event type; if the event type is also the same, we sort them by their y-coordinate. Finally, we iterate through the events and do the following:

1. If the event is the start of a belt, add the belt to the BST
2. If the event is a ball falling from a belt to the left, find the belt it hits first. We find the belt by searching the BST, since it contains all the belts that have had their start before, and have not ended yet.
3. If the event is a ball falling from a belt to the right, we do the same thing.
4. If the event is a dropped ball, we also do the same thing.
5. Finally, if the event is the end of a belt, we remove it from the BST

Here we can see why enumerating the events was important – when processing those with the same y-coordinate, it is crucial that we insert all the new belts first, and only remove those that have ended after we have processed all the balls. Because of the BST, all the events have the complexity of $\Theta(\log n)$, which leads to a final complexity of $\Theta(n \log n)$.

Solution by:

Name: **Nenad Božidarević**

School: Faculty of Computing

E-mail: nbozidarevic@gmail.com

Problem R1 08: Palindromes (ID: 9748)

Time Limit: 4.0 second

Memory Limit: 256 MB

Given a string, you keep swapping any two characters in the string randomly till the string becomes a palindrome. What is the expected number of swaps you will make?

Input

The first line contains the number of test cases T . Each of the next T lines contains a string each.

Output

Output T lines containing the answer for the corresponding test case. Print the answer rounded to exactly 4 decimal places.

Sample

input	output
4	0.0000
b	0.0000
bb	3.0000
abb	59.3380
cbaabbb	

Constraints $T \leq 10000$

The length of the string will be at most 8 characters.

The string will consist of only lower-case letters 'a'-'z'.

There will always be at least one palindrome which can be formed with the letters of the given string.

Solution:

This is a nice and adequate problem (possibly a bit on the harder side) that requires a basic knowledge of probability and good thinking. Given a string of length l , a **character swap** is defined by the positions of the two characters that we are swapping in it. Hence, there are $\binom{l}{2} = \frac{l(l-1)}{2}$ possible character swaps, and since each of them is equally probable, the probability of picking any particular swap is $P_{\text{swap}} = \frac{2}{l(l-1)}$.

Let $E(S)$ denote the **expected value** of the amount of random swaps needed to reach a palindrome, starting from string S . If S is a palindrome, we need make no further swaps, hence $E(S) = 0$ in that case. For a non-palindromic string, we can relate its expectation to the expectations of all other strings we can obtain from it after 1 swap. What we must note is that we are obliged to make one swap; after that, the expectation of the remaining number of moves can be equated to the expected remaining swap amount of the newly obtained string. This gives us the expression

$$E(S) = 1 + \sum_i P_{\text{swap}} \cdot E(S_i)$$

where i iterates over all the possible swaps in S , and S_i denotes the string obtained after performing the i^{th} swap. This can be converted into a system of linear equations where the unknown variables are the expected values for each string involved. We can solve this system in $O(n^3)$ time complexity using an algorithm such as **Gaussian elimination**.

The main problem here is that n can be too large for this algorithm to handle in time – in fact, it is equal to the amount of permutations of the string (i.e. $n = l!$). In order to reduce n to a more feasible value, a key observation must be made – there are many equivalencies between pairs of permutations. For example, the strings “bbaadfa” and “aabbcbg” have the same character distribution, only with different characters – it’s clear that their expected values are the same. Hence, we can reduce the amount of variables in our system drastically by **relabeling** the strings in some predetermined way – one of the simplest ways is to convert all the occurrences of the first distinct character in the string to ‘a’, the next one to ‘b’ and so forth. Another rather useful thing to consider is that solving a system of equations gives us a whole lot of additional solutions as well as the one we are currently looking for. As we have multiple test cases per input file, a completely reasonable optimization is to store these solutions for later use, by using a structure such as the C++ **map** (binary search tree).

Let $gauss[][]$ be the matrix representing the system of equations we need to solve (initialized to 0 and zero-indexed), and n the number of mutually non-equivalent permutations of the input string S . Now we can define our final algorithm in steps:

1. Relabel S .
2. Assign a unique index to every relabeled permutation of S .
3. Check if an equation system has already been solved for S . Output solution and continue to next test case if so.
4. Check if S is a palindrome. If so, output zero and continue.
5. For each permutation of S , S_i :
 - a. Relabel S_i and get its index, x .
 - b. Set $gauss[x][x] = 1$.
 - c. If S_i is a palindrome, go to next permutation.
 - d. Set $gauss[x][n] = 1$.
 - e. For each string S_j reachable from S_i in one swap:
 - i. Relabel S_j and get its index, y .
 - ii. $gauss[x][y] = gauss[x][y] - P_{swap}$.
6. Solve the system using Gaussian elimination.
7. Record all the solutions of the system for later use and output the required one.

Overall, the time complexity of generating the equation system is $O(l^2 \cdot l!)$ and the asymptotic complexity of solving the system of equations is $O(n^3)$. This gives us a total time complexity of $O(n^3 + l^2 \cdot l!)$ per test case, which is sufficient to pass the time limit. The space complexity of the solution is $O(n^2)$.

Solution by:

Name: **Petar Veličković**

School: *University of Cambridge*

E-mail: *pv273@cam.ac.uk*

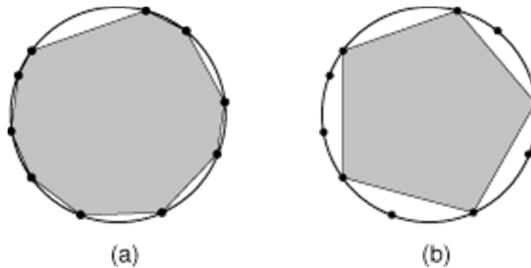
Problem R1 09: Shrinking Polygons (ID: 3415)

Time Limit: 1.0 second

Memory Limit: 256 MB

A polygon is said to be inscribed in a circle when all its vertices lie on that circle. In this problem you will be given a polygon inscribed in a circle, and you must determine the minimum number of vertices that should be removed to transform the given polygon into a regular polygon, i.e., a polygon that is equiangular (all angles are congruent) and equilateral (all edges have the same length).

When you remove a vertex v from a polygon you first remove the vertex and the edges connecting it to its adjacent vertices w_1 and w_2 , and then create a new edge connecting w_1 and w_2 . Figure (a) below illustrates a polygon inscribed in a circle, with ten vertices, and figure (b) shows a pentagon (regular polygon with five edges) formed by removing five vertices from the polygon in (a).



In this problem, we consider that any polygon must have at least three edges.

Input

The input contains several test cases. The first line of a test case contains one integer N indicating the number of vertices of the inscribed polygon ($3 \leq N \leq 104$). The second line contains N integers X_i separated by single spaces ($1 \leq X_i \leq 103$, for $0 \leq i \leq N - 1$). Each X_i represents the length of the arc defined in the inscribing circle, clockwise, by vertex i and vertex $(i + 1) \bmod N$. Remember that an arc is a segment of the circumference of a circle; do not mistake it for a chord, which is a line segment whose endpoints both lie on a circle.

The end of input is indicated by a line containing only one zero.

Output

For each test case in the input, your program must print a single line, containing the minimum number of vertices that must be removed from the given polygon to form a regular polygon. If it is not possible to form a regular polygon, the line must contain only the value -1.

Sample

input	output
3	0
1000 1000 1000	2
6	-1
1 2 3 1 2 3	5
3	
1 1 2	
10	
10 40 20 30 30 10 10 50 24 26	
0	

Solution:

Let's reformulate this task a little bit. Given a circular array of integers A_i , our task is to partition this array into a maximal number of blocks consisting of consecutive elements in the given array, such that the sum of the elements in each block is the same.

Since the sum in each block should be the same, we can note that this sum has to divide the sum of all the elements in the array S . Now, for some k that divides S , we need to check if it is possible to partition our array into S/k blocks with sums equal to k . One way to do this is to start the first block with element i , $0 \leq i \leq n - 1$, and keep adding elements $\{(i + 1) \bmod n, (i + 2) \bmod n, \dots\}$ to this block until block sum is k . If the value k is skipped (before adding some element, the sum of the block is less than k , and after adding the element it is larger) or the sum can't reach k (because all elements have been taken), we can't do the partition when the first block starts with element i . Otherwise, we start a new block and keep doing this until all the elements are taken. At the end, we just take the maximum of all k -s for which it is possible to partition the input array.

Time complexity of this solution is $O(\text{divNum} \cdot n \cdot n)$, where divNum is the number of divisors of S .

But we can speed this up. We can note that if it is not possible to partition the array when the first block starts with element i and $A_i + A_{i+1} + \dots + A_{j-1} \bmod k = 0$ then it is not possible to partition the array when the first block starts with element j either. Now, with this in mind, let d_i represent the sum of the elements at the end of the array A that don't get in any block with the algorithm above (without going circular). How do we calculate d_i ? Well, we can do that by finding index j , such that $A_i + A_{i+1} + \dots + A_{j-1} = k$, and then using the previous observation we have $d_i = d_j$. If such j doesn't exist, let $d_i = A_i + A_{i+1} + \dots + A_{n-1}$. We fill the array d by starting at the end of the array A and moving towards the start. Finding j can be easily done using binary search, but it can be even done just using one pointer and moving it as we move our index i , keeping track of the sum between i and this pointer $\leq k$. After this we only need to check if $A_0 + A_1 + \dots + A_{i-1} + d_i = k$ or $d_0 = 0$. If either of these two conditions hold, we can partition A into S/k blocks with sums equal to k .

Time complexity of this solution is $O(\text{divNum} \cdot n)$.

Solution by:

Name: Boris Grubić

School: University of Cambridge

E-mail: borisgrubic@gmail.com

Problem R1 10: [CH] Tetris AI (ID: 758)

Time Limit: 30.0 second

Memory Limit: 256 MB

In the very heart of a well known producer of microelectronic products, a mobile phone with a built in game of Network Tetris is being prepared for release. The owners of such mobile phones can arrange duels when at a small distance from each other. Data transmission between players is carried out using the Bluetooth protocol.


However -- now we are coming to the point -- it sometimes happens that there may be no other similar phone nearby and the player may need to play alone. For this purpose it is necessary to write a computer player (AI) with a very hard difficulty level.


The rules of Network Tetris are pretty simple :

The game has two playing fields, each with the rules of standard Tetris: figures of 4 blocks keep falling from the top of the field, and have to be placed in such a way as to form horizontal lines. Once a line is filled up, it is removed and all lines above it are appropriately shifted downwards. There is however one difference with respect to standard Tetris -- a player receives additional penalty lines as soon as his opponent clears a line. The game is over when one of players fills his own field, either on his own or with his opponent's help, to such an extent, that the next figure cannot fully enter the field.


The width of the field is 10, and the height of field is 20. There are 6 types of figures in the game:


I (1) - 

L (2) - 

J (3) - 

Z (4) - 

S (5) - 

O (6) - 



It is your task to write a bot which starts with an empty field and, knowing the sequence of figures dropping on its field, plays in such a way as to do as much harm as possible to the opponent.

Input

t – number of test cases [$t \leq 150$], then t tests follow.

Each test case starts with integer N equal to the number of figures which drop onto the field [$10 \leq N \leq 50000$]. Then N integers follow, numbered from 1 to 6 – denoting one of the figures, in the same position as they appear in the pictures. (Look at numbers in parenthesis).

Output

For each test you should output line case 1 Y if you wish to solve this test case or case 1 N otherwise. If you output Y , then exactly N lines must follow. Each of them should contain exactly two integers: A and X , where A is the clockwise angle of rotation in the given move, numbered from 0 to 3 (0 - 0 degrees, 1 - 90 degrees, 2 - 180 degrees, 3 - 270 degrees), while X is the horizontal coordinate of leftmost cube of the figure [$1 \leq$

$x \leq 10$]. The i -th figure at input corresponds to the i -th figure at output. If the figure falls outside the field or the parameters have incorrect value, or any falling figure stops with at least one cube in a line of number larger than 20, then the solution will be judged as Wrong Answer.

Score

The score will be equal to the total number of cleared lines, taken over all test cases. For one cleared line your solution will receive 1 point, for two simultaneously cleared lines – 5 points, for three simultaneously cleared lines – 15 points, for four simultaneously cleared lines – 30 points.

Sample

input	output
1	case 1 Y
14	0 1
3	0 8
2	0 1
4	0 8
5	1 7
3	3 3
2	1 5
1	0 7
6	0 9
6	0 1
1	1 6
1	1 6
4	0 1
1	1 4
5	

Score :

score = 30 + 1 = 31

Solution:

Tetris is a very famous classic in the world of video games, released back in 1984. Artificial intelligence for tetris is not an unknown topic. Actually, a lot of work on Tetris AI can be found on the internet. While this challenge problem is not the exact version of the classical Tetris game, resources that can be found are a good starting point in attacking the scoreboard and making the best AI.

As stated, there are some differences between **[CH] Tetris AI** and the classical game of Tetris. There are only 6 types of figures in this problem, while the original has 7 – the very useful *T* figure is missing. This makes the game a bit harder. However, there is an advantage that we are presented with the whole sequence of figures in the game here. Sadly, it turns out it's not a significant advantage. The main reason for that, and the last important difference, is the time limit. This problem has a very strict time limit, which makes it almost impossible to use any advanced AI and algorithms.

Obviously, the first thing that comes to our mind is the brute force approach in solving the problem and we are going to use that. For the next figure, we are going to try all the possible rotations and all the columns and pick those that seem *the best*. This is the greedy approach. Since we know the whole sequence of figures, we can look ahead a certain number of figures and try all the combinations of rotations and columns of several figures. This will, however, slow down our program significantly. The number of combinations for one figure is not more than $10 \cdot 4$ (number of columns times number of different rotations), but the number of combinations for k figures can be up to $(10 \cdot 4)^k$. It turns out that we don't have to test more than one figure, and that the full greedy approach of trying all the combinations of the current figure only is good enough to get the top score.

So, this is the basic algorithm of our program. But, we haven't defined what the best move is. We can score

the move by calculating certain features of the move. Let's define a score of the move as

$$score = weight_1 \cdot feature_1 + weight_2 \cdot feature_2 + \dots + weight_n \cdot feature_n$$

where n is the number of features. Weights are real numbers from the interval $[-1, 1]$, while a feature could be any real number. For the best move, we will take the one with the lowest score.

Now it's important to make a good list of features we will use to score our move. It seems obvious that the more (useful) features we use, we have a better scoring system.

Here is the list of features we could use:

- number of lines cleared by the move
- number of holes (empty cells which have a filled cell above them in the same column)
- highest column
- average height of all blocks
- number of edges of the current figure touching other figures or walls
- row transitions (empty cell adjacent to filled cell in the same row)
- column transitions (empty cell adjacent to filled cell in the same column)
- number of wells (succession of empty cells such that their left and right cells are both filled)
- etc.

Although calculating any of these features does not seem a very time consuming task, due to the very strict time limit, it is not possible to use large number of features. Actually, for a good AI it is enough to use just a few features. One choice of features that gave good results was a set of only three features: number of edges, number of holes and average height of all blocks. Even with the three features only, it was very important to optimize the program everywhere, skipping all the unnecessary calculations – not doing anything twice, only working on the relevant part of the field and not going over the whole field every time...

When we have chosen a set of features, we are still left with figuring out the best weight for each feature. Obviously, weight corresponding to some features should be negative (e. g. number of lines cleared), while for some it should be positive (e. g. number of holes). So, one way of figuring out the weights is using our human intuition and a try a few different combinations to see which one works the best – which one gets us the highest score in the whole test set. But, there is a better way of finding the best combination of weights!

Brute force search is out of the question, but we can use one of the optimization techniques to find a good combination of weights for the chosen set of features. One of the optimization techniques that are suited for this kind of thing are **genetic algorithms** (http://en.wikipedia.org/wiki/Genetic_algorithm). The basic idea is to generate a few Tetris games (sequences of figures) and run the genetic algorithm on this set. The genetic algorithm should look for the combination of weights that gets the highest score in these games. We will then code the resulting weights into our program that will be submitted to SPOJ.

This is a simple greedy approach that yields great results on this problem. Also, there is one simple heuristic that can easily be combined with this approach for greater results. We can leave the leftmost (or rightmost) column empty until the height of all the other columns is greater than 3. By leaving it empty, we make room for the *I* figure. Since clearing multiple lines at once gives us more points, this is a good way to score even 30 points by clearing four lines!

All of this combined makes a program good enough for the top of the Tetris AI scoreboard. Of course, there is room for improvements, like using a different set of features or adding some other simple heuristics that will increase the scoring potential of the AI even more. Those who haven't tried to solve this problem seriously should definitely give it a try, as it is a great introduction to the world of AI.

Solution by:

*Name: **Vanja Petrović Tanković***

School: Faculty of Computing

E-mail: vpetrovictankovic@gmail.com

Problem R2 01: Avoiding SOS Grids (ID: 6999)

Time Limit: 5.0 second

Memory Limit: 256 MB

You are given a grid having N rows and M columns. Some squares are contain letters 'S' or 'O', whereas the other squares are unfilled. A filled grid is called "Avoiding SOS" if there is no occurrence of the string "SOS" in the grid either vertically or horizontally. In how many ways can the grid be completed by filling the unfilled squares with either 'S' or 'O' such that the resultant grid is "Avoiding SOS" ?

Input

The first line contains T the number of test cases. T test cases follow. The first line for the test case contains N and M , the number of rows and columns in the grid respectively. N lines follow, each containing M characters. The j th character in the i th line is a '.' if the corresponding square in the grid is unfilled, otherwise it contains either the letter 'S' or the letter 'O'. A blank line separates two test cases.

Output

Output T lines, one for each test case, containing the desired answer for the corresponding test case. Output each result modulo 1000000007.

Sample

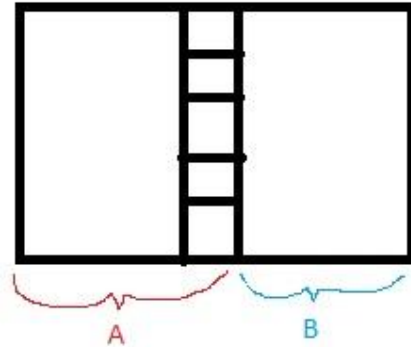
input	output
5	49
2 3	12
...	9
...	0
	1
1 4	
....	
3 3	
.O.	
S.S	
.O.	
1 3	
SOS	
1 3	
SOO	

Constraints $1 \leq T \leq 100$ $1 \leq N, M \leq 8$

Solution:

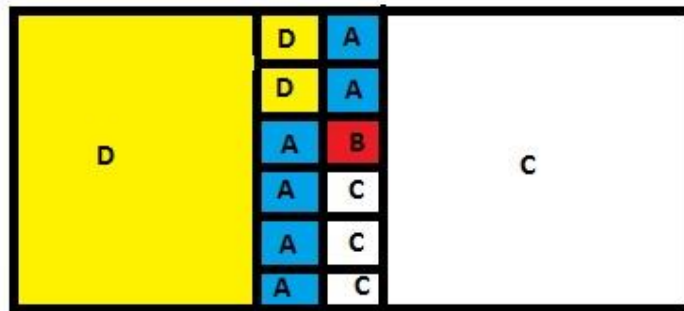
In short, this problem is solved with dynamic programming on profile. But the approach with complexity $O(3^{2 \cdot N} \cdot N)$ will probably time out and we should implement it with complexity $O(3^N \cdot N^2)$.

The $O(3^{2 \cdot N} \cdot N)$ approach works as follows:



We split the board into two parts – A and B. For part A we know what we have filled, and part B is unfilled. Notice that the string “SOS” consists of 3 characters, so we do not need to know everything in part A. We can group all possible A-parts by their length and the states of the cells in the last filled column. There can be three interesting states for a cell in the last column – it is an S, an O with O before and an O with S before. So the state of the dynamic programming is $(length, mask\ of\ last\ column)$. From $(length, mask\ of\ last\ column)$ we can make the transition to $(length + 1, new\ mask\ of\ last\ column)$ by iterating through all possible masks for the next column and checking for compatibility with the no-SOS rule. The number of states is $O(3^N \cdot N)$, but we need $O(3^N)$ steps to transition to a new state, so this makes the complexity $O(3^{2 \cdot N} \cdot N)$. The last N can be improved to $\log(N)$, but this will still probably time out.

To make the solution faster we can notice that iterating through all new masks is expensive. Instead we can modify the mask so that we can iterate only one cell at a time.



As before, the cells in A and D are already filled, and in B and C – unfilled. The mask is now broken at cell B and it consists again of three states – S, O with O before, O with S before. The state of the dynamic programming is $(mask, coordinates\ of\ cell\ B)$, so there are $O(3^N \cdot N^2)$ states. The good thing with such a mask is that a transition, as we wanted, takes constant time – we just have to try putting O and S at B and check for compatibility. So we have solved the problem.

Note that when cell B is at the top row the mask becomes a whole column, which might look strange. Also, to make implementation easier, we can notice that the row with index (-1) can be filled with Os, since starting patterns O-O and O-S are neutral in regards to the overall solution.

Solution by:

Name: Vladislav Haralampiev

School: Sofia University

E-mail: vladislav_haralampiev@abv.bg

Problem R2 02: High and Low (ID: 12210)

Time Limit: 2.0 second

Memory Limit: 256 MB

A student is always bored during his math classes. As the teacher won't let him sleep, he finds something else to do: try to find patterns in the numbers written on the board! Currently, he's spending his time trying to find Hi&Lo subsequences.

Intuitively, a Hi&Lo sequence is any sequence that has consecutive differences with opposite signs, that is, if the first number is greater than the second, then the second is smaller than the third, the third is greater than the fourth, and so on.

Formally, let $x[1], x[2], \dots, x[n]$ be the numbers written on the board. A Hi&Lo subsequence of length k that only uses elements from A to B is a sequence of indices a_1, a_2, \dots, a_k such that:

1. $B \geq a_k > \dots > a_2 > a_1 \geq A$
2. $x[a_i] - x[a_i - 1] \neq 0, \text{ for } 1 < i \leq k$
3. $(x[a_i] - x[a_i - 1])(x[a_i + 1] - x[a_i]) < 0, \text{ for } 1 < i < k$

Note that every sequence with only one element is a Hi&Lo sequence.

However, as the amount of numbers increases, finding a big subsequence is getting harder and harder, so he asked you to create a program to help him and quickly find the largest Hi&Lo subsequence in a given interval.

Input

The input contains several test cases. A test case begins with a line containing integers N ($1 \leq N \leq 100000$) and M ($1 \leq M \leq 10000$), separated by spaces. On the second line there are N positive integers, the initial state of the board. M lines follow, each with an instruction. Instructions can be of two kinds:

- 1) $q\ A\ B$: Print the length of the longest high & low subsequence only using elements in positions from A to B , inclusive. You may assume that $1 \leq A \leq B \leq N$.
- 2) $m\ A\ B$: Modify the A th element of the sequence to the positive integer B . You may assume that $1 \leq A \leq N$.

No number on the board will ever exceed 10^9 . There is a blank line after each test case. The last test case is followed by a line containing two zeros.

Output

For each instruction of type 1, print a line containing an integer, the answer to the query. After each test, print a blank line.

Sample

input	output
5 7	2
1 2 3 4 5	3
q 2 4	2
m 3 1	1
q 2 4	
m 3 2	
q 2 4	
m 4 2	

q 2 4	
0 0	

Solution:

At first we will assume that all numbers in the array are different.

Let's call an element the "peak" if it is at the start or end of an array, or if it is greater or smaller than both of its neighbors.

Lemma 1: A sub-sequence of optimal length can consist only of peaks on a given interval.

The proof is left to the reader as an exercise.

It's easy to see that we can form a valid sub-sequence from taking all peaks on an interval.

From the **Lemma 1** we can see that it is a sub-sequence of optimal length.

If there were no updates then we could find all peaks in linear time and store them in some array S , where S_i would be the number of peaks before the i -th element.

This way, we could answer all queries in $O(1)$ with $O(N)$ precalculation.

We could process our updates by simply updating our array S , but that would take $O(N)$ time for each update giving $O(N \cdot Q)$ total time, which is too slow.

Because of that we can use a segment tree (or a similar data structure) to compensate for updates.

Given that we only need the number of peaks on an interval in order to answer the query, it is intuitive that we need to keep the number of peaks in a node of a segment tree. When joining intervals that two nodes in a segment tree cover, we can see that the number of peaks cannot change drastically. Only nodes at ends of intervals can stop being peaks. Therefore, beside the number of peaks, a node of the segment tree will also store the first element, last element and the relation between the first two elements on an interval (if they exist).

It is easy to see now that the case where array elements can be equal is almost the same with a few extra checks.

The time consumed for each query is $O(\log N)$ and initializing the segment tree takes only $O(N)$. Therefore, overall time complexity is $O(N + Q \cdot \log N)$ and memory complexity is $O(N)$.

Solution by:

Name: **Marko Stanković**

School: Gimnazija "Svetozar Markovic" Nis

E-mail: marko.stankovic996@gmail.com

Problem R2 04: Tower Game (Hard) (ID: 7857)

Time Limit: 1.0 second

Memory Limit: 256 MB

Daniel is building towers out of blocks. He has many black and white blocks. He has built n towers out of those. Now he suggests Max playing the following game. Black block will belong to Daniel and white blocks will be Max's blocks. During his turn the player can take any of his blocks from any tower and remove it and all the blocks above it. As usual the player who can't make the move loses. Daniels make the first move. Determine who will win if both players play optimally.

Input

The input starts with number t - the amount of test cases. The first line of each test is number n - the number of towers. Then n strings follow. Each string is formed of 'B' and 'W' characters, where 'B' means black block and 'W' - white block. Each string describes one tower from bottom to top.

Output

For each test case print 'Win' if Daniel wins and 'Loss' if Max wins given both players play optimally.

Sample

input	output
1 5 BBWWB BWBB BB WWW WB	Win

Constraints $1 \leq t \leq 20$ $1 \leq n \leq 1000$

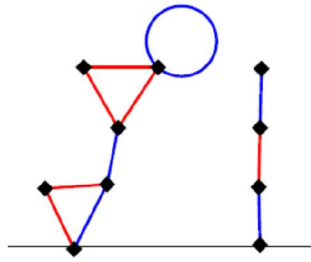
String consist of no less than 1 character and no more than 1000 characters.

Solution:

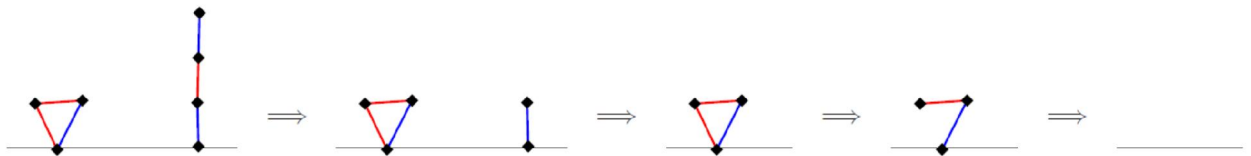
This task requires solid knowledge of **combinational game theory**.

Red-Blue Hackenbush is a partisan version of the game **Hackenbush**. A partisan game is not impartial, which means that there are certain moves that are available to one player but not to the other. A Red-Blue Hackenbush position is a set of points connected by line segments that are colored either red or blue. Some of those points lie on the ground line. A segment is connected to the ground if at least one of its endpoints lie on the ground, or if the segment is connected to another segment that is connected to the ground. There are two players in Red-Blue Hackenbush called Blue and Red who alternate moves. Blue can only cut blue segments and Red can only cut red segments. When a player cuts a segment the segment is removed together with any other segment that is no longer connected to the ground. The first player unable to move loses according to the **normal play convention**.

Below is a sample game of Red-Blue Hackenbush. Let's assume that Blue is to make the first move.



Blue's first move is to cut the blue segment that connects the two triangular structures on the left. When he cuts that segment the triangular structure above it and the blue self-loop will also be removed and we can see the result of that move on the picture below along with a couple more moves.



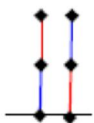
We can see that Red made a mistake in his last move. If he had chosen to remove the red segment parallel to the ground line he would have won. However, that is not the only non-optimal move in this game, there are a few more.

Let's make a few simple observations about the game:

- If there are no edges, then the player whose turn it is to move loses.
- If some connected portion of the graph that is rooted to the ground contains only n edges of a single color then by cutting those edges optimally the player of that color can make n moves on that portion.
- If the red and blue edges do not interact – on other words, if cutting a red edge can never cause a blue edge to be removed and vice-versa, then the result of the game where both players play optimally with m such red edges and n such blue edges depends only on the value $n - m$. If $n - m$ is positive, Blue will win. If $n - m$ is negative, Red will win. If $n - m$ is equal to zero, the second player will win.
- A lot of Hackenbush positions can be broken up into components which have no edges or vertices in common, except possibly for the vertex on the ground. If all components are purely red or blue, we can assign a value of $+n$ to a component with n edges if it is entirely blue or $-n$ if it is entirely red. If we sum the values of all components of a position we will have a value of that position and if that value is positive Blue will win, if it's negative Red will win and if it's equal to zero, the second player will win.

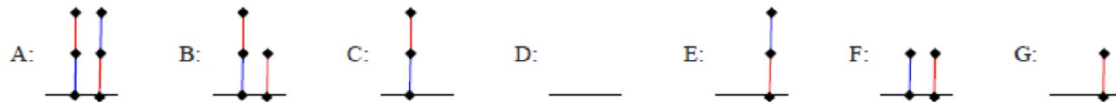
Although the last observation we made is a simple observation, the following is not: it turns out that we can assign numerical values to components where the edges do interact, and those values can be added to give the game a value. From now on, we will always associate positive values with positions where Blue has an advantage and negative values with positions where Red has an advantage. A game with zero value will always yield a win for the second player.

In any game that has no ties and where there is a finite number of possible moves, there is always a method to determine the optimal strategy for each player. Let's analyze the following Hackenbush game:

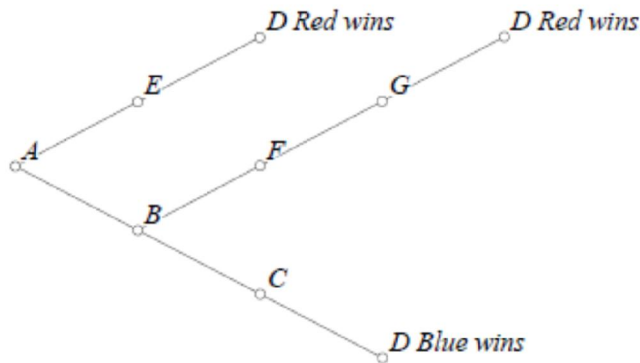


Let's assume that Blue makes the first move. There are 7 possible positions that can be reached, in one or

more moves, from the initial position including the initial position.



We can now draw the following diagram for the game which will tell us how we can advance from one position to another.



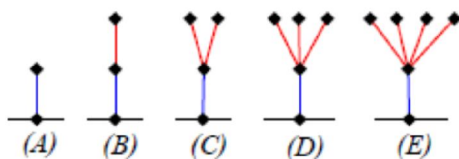
From the graph we can see that no matter what move blue chooses Red will always have a winning strategy and will eventually win, if he plays optimally.

Let's make a few more observations:

- If a blue edge is added to any position in any way, the position becomes better for Blue, and vice-versa. This one is pretty obvious, since having an extra move can't hurt.
- Symmetry can often be helpful in analyzing the position. If a position can be divided into two symmetric parts where the structures are identical, but the colors reversed, then this is a position, where second player wins by playing optimally since he can just mirror the moves of the first player. Examples of those positions are *A* and *F* on the picture above.

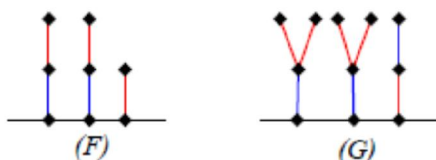
We have shown that integers can be assigned to certain positions. An interesting thing is that fractional values can also be assigned to certain positions.

Let's take a look at the following components:



In component A Blue has exactly one move, and this component should be assigned a value of 1. In all other components, Blue can win whether he starts or not, but each successive component to the right is less desirable for Blue since Red has more and more options. In a position that contains a component like *E* above, if Blue needs to make moves in other components because they are even more critical, Red has up to four "free" moves. So if the value of component *A* is 1 what values should we assign to the remaining components?

Let's take a look at the position *F* below:

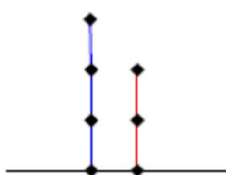


If we analyze position F we can see that its value is 0, which means that with optimal play by both players whoever plays first will lose. Since the position F consists of two components B and a single red segment (with a value of -1) it's highly reasonable to assign a value of $1/2$ to component B .

Now consider position G . The rightmost component of G is the same as B , but with the colors reversed. That component most likely has a value of $-1/2$. If we analyze position G , we can see that it has zero value. That means that position C should probably have a value of $1/4$ assigned to it.

It will also turn out that reasonable values for components D and E are $1/8$ and $1/16$, respectively.

Let's consider a very simple game:



There are two independent components that are completely blue or completely red. For a game this simple, the best moves for each players are obvious: always hack off the uppermost segment. Since Blue has 3 moves and Red has 2 moves the reasonable value for this game would be $3 - 2 = 1$. Blue can move from this position to positions with values of 0, -1 and -2 . Red can move from this position to positions with values of 2 and 3. Remember that when Red makes a cut, the situation becomes worse for him, since his number of options reduces. That means that the game becomes better for Blue and the game value must increase. However, if Blue makes a move then the game becomes better for Red and the game value decreases. Let's say that Blue can reach games with values B_1, B_2, \dots, B_n in one move from the current game and that Red can reach games with values R_1, R_2, \dots, R_m in one move from the current game. We can come to a conclusion that the true value of the game V must be larger than all of the numbers B_i and smaller than all of the numbers R_j . From now on we will use the following form to denote games:

$$V = \{B_1, B_2, \dots, B_n | R_1, R_2, \dots, R_m\}$$

Since we know that the game value will be a number that is larger than all of the numbers B_i and smaller than all of the numbers R_j , we can discard all of the numbers B_i except the largest one and all of the numbers R_j except the smallest one, since those moves aren't optimal for the players. For the sample game above, there are three possible Blue moves and two possible Red moves with the values stated above, and we can write the value of the game as:

$$\{-2, -1, 0 | 2, 3\} = \{0 | 2\}$$

The aforementioned number is called a surreal number. Surreal numbers were introduced in Donald E. Knuth's fiction book *Surreal Numbers: How two ex-students turned on to pure mathematics and found total happiness* and the full theory behind them was developed by John H. Conway in his book *On Numbers and Games*.

The important thing about surreal numbers is that each Red-Blue Hackenbush game has a surreal number assigned to it. There are a couple of theorems about surreal numbers which are very useful in analyzing the Red-Blue Hackenbush game. We will only list those theorems here but we will not prove them.

- **The Simplicity Rule** – Let b be the largest value of any position to which Blue can move. Let r be the smallest value of any position to which Red can move. The condition $b < r$ will always hold.

- If there is an integer n satisfying $b < n < r$, then value of the game is the closest such integer to 0.
- Otherwise value of the game is the unique rational number x satisfying $b < x < r$ whose denominator is the smallest possible power of 2.
- If a surreal number corresponds to a standard rational number (which is always the case in Red-Blue Hackenbush), then the sum of surreal numbers behaves the same as sum of rational numbers, and the game represented by the combination of two components with values of V_1 and V_2 will have a value of $V_1 + V_2$.

Examples of The Simplicity Rule:

b	r	x
$2\frac{3}{4}$	$6\frac{1}{2}$	3
-5	$2\frac{5}{8}$	0
0	1	$\frac{1}{2}$
$\frac{1}{4}$	$\frac{5}{16}$	$\frac{9}{32}$
$\frac{1}{4}$	$\frac{7}{16}$	$\frac{3}{8}$
$-2\frac{7}{8}$	$-2\frac{3}{32}$	$-2\frac{1}{2}$

Now we know how to find the value of the game if we know the values of all of its sub-games.

In Hackenbush there are some special positions like trees and stalks. A “tree” is a connected position where there is a unique path from any node to the “ground” node. In other words, there are no loops in a tree. A “stalk” is a very special kind of tree where there are no branches, which means that every node has at most two segments connected to it.

There exists a simple method to calculate the value of a Red-Blue Hackenbush stalk devised by Elwyn R. Berlekamp. Let’s assume that the base segment of the stalk is blue. If it’s red we can just negate all of the values.

- Count the number of blue segments that are on the continuous path from the ground node. If there are n of them, start with the number n .
- For each new segment going up, assign the value of that segment to be half of the one below it, and add it to the sum if it’s blue, or subtract it if it’s red.
- When you reach the top of the stalk you will have the final value of the stalk.

For example, the value of the stalk whose segments are (starting from the ground) BBBRRBRRBR is:

$$3 - 1/2 - 1/4 + 1/8 - 1/16 - 1/32 + 1/64 - 1/128 = 293/128$$

Now let’s go back to our task. The following observation is trivial. The game from the task is the same as Red-Blue Hackenbush, but with different colors (black and white). We can use the knowledge learned about Red-Blue Hackenbush to solve this task. Let Daniel be Blue and Max be Red and let each tower be a component in a Red-Blue Hackenbush position. We can use Berlekamp’s rule to calculate the value of the game and use that value to determine the winner of the game. Now the only thing remaining is the implementation. We need to find the best way to sum dyadic fractions, numbers of the form $a/2^b$. In this case the lowest possible fraction in our sum is $1/2^{999}$, which is much smaller than the machine epsilon value for double precision floating point numbers – the smallest possible representable number in that standard. There are a couple of ways to solve this problem:

- Make your own number class for big integers (integers with many digits, in our case thousands of digits), or use the `java.math.BigInteger` class and then make your own class for fractions whose numerators and denominators are big integers and then just sum

the values of each tower and get the value of the game and then determine the winner from the sign of that value.

- Since we are only interested in the sign of the value, not the value itself, there exists a simpler method to determine the sign of the value. Make a polynomial of the following form: $a_0 * 2^0 + a_1 * 2^{-1} + a_2 * 2^{-2} + a_3 * 2^{-3} + \dots + a_{999} * 2^{-999}$ and then sweep that polynomial from the right to the left and make adjustments to the coefficients such that every coefficient, except possibly a_0 , belongs to the set $\{-1, 0, 1\}$. In order to do that we should do a loop from 999 to 0 and change each a_i to $a_i \bmod 2$, which represents the remainder of division of a_i by 2 and then if a_i is positive add $a_i/2$ to the number a_{i-1} and if the number a_i is negative we should add $-|a_i + 1|/2$ to the number a_{i-1} . Now, in order to find the sign of the value it's enough to sweep from the left to the right and find the sign of the first coefficient which is not equal to zero.

Solution by:

Name: **Aleksandar Ivanović**

School: University of Belgrade, School of Electrical Engineering

E-mail: aleksandar.ivanovic.94@gmail.com

Problem R2 05: Fibonacci recursive sequences (hard) (ID: 12009)

Time Limit: 40.0 second

Memory Limit: 256 MB

Leo searched for a new fib-like problem, and ...

it's not a fib-like problem that he found !!! Here it is.

Let FIB the Fibonacci function :

$$FIB(0) = 0 ; FIB(1) = 1$$

and

$$\text{for } N \geq 2 \quad FIB(N) = FIB(N - 1) + FIB(N - 2)$$

Example : we have $FIB(6) = 8$, and $FIB(8) = 21$.

Let $F(K, N)$ a new function:

$$F(0, N) = N \text{ for all integers } N.$$

$$F(K, N) = F(K - 1, FIB(N)) \text{ for } K > 0 \text{ and all integers } N.$$

$$\text{Example : } F(2, 6) = F(1, FIB(6)) = F(0, FIB(FIB(6))) = FIB(FIB(6)) = FIB(8) = 21$$

Input

The input begins with the number T of test cases in a single line.

In each of the next T lines there are three integers: K, N, M .

Output

For each test case, print $F(K, N)$,

as the answer could not fit in a 64bit container,

give your answer modulo M .

Sample

input	output
3	5
4 5 1000	1
3 4 1000	21
2 6 1000	

Constraints

$$1 \leq T \leq 10^3$$

$$0 \leq K \leq 10^{18}$$

$$0 \leq N \leq 10^{18}$$

$$2 \leq M \leq 10^{18}$$

K, N, M are uniform randomly chosen.

Solution:

The problem is about the Fibonacci sequence and, in order to find an efficient algorithm, some nontrivial properties of this famous sequence are required. Denote the n -th Fibonacci number by $F(n)$ and let $F^k(n) =$

$F(F(\dots F(n) \dots))$ (k times). We need to calculate $F^k(n) \bmod m$, for given k, n and m . These numbers can be as large as 10^{18} and we can have up to 1000 such triplets in a single input; time limit per input is 40s which means that we must be able to solve the single triplet (k, m, n) in less than 0.05s on average. However, task statement says that all these triplets are generated **uniformly at random** – very useful information which suggests that we need an algorithm which has good **average** complexity and not necessarily good worst-case complexity (since there aren't any manually-made tricky test cases).

For starters, let us consider the simpler variants of the problem with small values of k . For $k = 1$ we only need to calculate $F(n) \bmod m$, which is well-known problem and can be done in $O(\log n)$. One possible way is **fast matrix exponentiation by squaring**:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}$$

and the other one is by applying recurrent formulas:

$$F(2n-1) = F(n)^2 + F(n-1)^2,$$

$$F(2n) = (2F(n-1) + F(n))F(n).$$

Of course, all calculations are done modulo m .

The case $k = 2$ is much more complicated. We need to calculate $F(F(n)) \bmod m$, but we cannot do it by calculating $n' = F(n) \bmod m$ and $F(n') \bmod m$, since $F(F(n) \bmod m) \bmod m \neq F(F(n)) \bmod m$ in general (take, for example, $n = 6, m = 4$). Indeed, $F(n)$ is now playing a role of **an index** in the Fibonacci sequence and it seems that we need that exact value (and not $F(n) \bmod m$). However, $F(n)$ is exponential in n (it has $\Theta(n)$ digits) and for large n it cannot be computed fast. What now?

The solution lies in the fact that for any fixed number m , sequence $\{F(n) \bmod m\}_{n \in \mathbb{N}_0}$ is **periodic**. Indeed, since every element of this sequence lies in $[0, m-1]$, the number of distinct pairs of consecutive elements is at most m^2 ; it follows that there exists indices i and j such that $F(i) \bmod m = F(j) \bmod m$ and $F(i+1) \bmod m = F(j+1) \bmod m$. Now, using the recurrent formula we see that $F(i+2) \equiv F(i) + F(i+1) \equiv F(j) + F(j+1) \equiv F(j+2) \pmod{m}$ and so on. Formula can be also applied “backwards” ($F(i-1) = F(i+1) - F(i)$) to show that pre-period length is 0. The (least) period of the sequence $\{F(n) \bmod m\}_{n \in \mathbb{N}_0}$ is called the **m -th Pisano period** and is denoted by $\pi(m)$. Here are examples for $m = 3$ and $m = 4$ (periods are underlined):

$F(n) \bmod 3$: 0, 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, 2, 0, 2, 2, 1, ...

$F(n) \bmod 4$: 0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1, 0, 1, 1, 2, 3, 1, ...

We can see that $\pi(3) = 8$ and $\pi(4) = 6$. For now, let us suppose that we can efficiently compute $\pi(m)$ and that $\pi(m)$ fits into 64 bits (we will later see how and why). By the definition of the period, we have $F(x) \bmod m = F(x \pm \pi(m)) \bmod m$, and, in particular, $F(x) \bmod m = F(x \bmod \pi(m)) \bmod m$ (i.e. we only need first period-block). For $x = F(n)$ we get

$$F(F(n)) \bmod m = F(F(n) \bmod \pi(m)) \bmod m,$$

which solves our problem for $k = 2$ because we can compute $x = F(n) \bmod \pi(m)$ in $O(\log n)$ and $F(x) \bmod m$ in $O(\log \pi(m))$ since $x < \pi(m)$.

Same reasoning can be applied for $k = 3$. We need $F(F(F(n))) \bmod m$ and by previous case it suffices to calculate $x = F(F(n)) \bmod \pi(m)$ and then $F(x) \bmod m$; this is exactly the previous case, except we have

$\pi(m)$ instead of m . Therefore

$$F(F(F(n))) \bmod m = F(F(F(n) \bmod \pi(\pi(m))) \bmod \pi(m)) \bmod m.$$

Now, denote by $\pi^k(m) = \pi(\pi(\dots\pi(m)\dots))$ (k times). Then, for an arbitrary $k \in \mathbb{N}$, we have

$$F^k(n) \bmod m = F(F(\dots F(n) \bmod \pi^{k-1}(m) \dots) \bmod \pi(m)) \bmod m$$

which can be solved in $O(\log n + \sum_{i=1}^{k-1} \log \pi^i(m))$. This is a significant progress, but we must do better since k can be up to 10^{18} .

We will again use the properties of Pisano periods. Suppose (again, we will elaborate later) that for our m , there exists an integer c_m such that $\pi^{c_m}(m) = \pi^{c_m+1}(m)$ i.e. the sequence $m, \pi(m), \pi^2(m), \dots, \pi^i(m), \dots$ becomes constant from the index c_m onwards (the constant value is obviously $\pi^{c_m}(m) = M \in \mathbb{N}$ with property $\pi(M) = M$). Then, we can calculate the expressions

$$x = F(F(\dots F(n) \bmod M \dots) \bmod M) \bmod M$$

(with $k - 1 - c_m$ mod operations) and

$$F(F(\dots F(x) \bmod \pi^{c_m}(m) \dots) \bmod \pi(m)) \bmod m$$

separately. First expression can be calculated more efficiently since we always have the same modulo: let $a_1 = F(n) \bmod M$ and $a_i = F(a_{i-1}) \bmod M$ for $i > 1$ (we have $x = a_{k-1-c_m}$). Since $0 \leq a_i < M$ for all i and since a_i depends only on a_{i-1} , we are bound to get into the cycle i.e. the sequence a is periodic (possibly with some pre-period). Let C be the smallest integer such that $a_C = a_i$ for some $i < C$. Then we have pre-period of length $i - 1$ and period (cycle) of length $C - i$ starting at index i . Now it suffices to calculate and memorize the first C element of the sequence a and the index i ; the calculation of $x = a_{k-1-c_m}$ reduces to finding the position of index $k - 1 - c_m$ in the cycle which is $O(1)$.

Let us summarize: if $P(x)$ is the complexity of calculating $\pi(x)$, then the total complexity of our approach for a single triple (k, m, n) is

$$O\left(C \cdot \log M + \sum_{i=1}^{c_m} P(\pi^i(m)) + \sum_{i=1}^{c_m} \log \pi^i(m)\right)$$

where the first term corresponds to calculation of the first C elements of sequence a (standard logarithmic Fibonacci number calculation and fact that $a_i < M$), the second term to the calculation of all necessary Pisano periods and the third term again for Fibonacci calculation in the second expression. If the numbers C and c_m are small enough and we can calculate Pisano periods fast enough, the problem is solved.

In order to back up our previous assumptions regarding $\pi(m)$ (which we took for granted) and to estimate C and c_m , we will need more theoretic knowledge about Pisano period. Here we list Pisano period facts (without proofs) which are needed for analysis and calculation of $\pi(m)$.

Pisano period facts:

1. For each $m \in \mathbb{N}$, $\pi(m) \leq 6m$ with equality iff $m = 2 \cdot 5^k$ for $k \in \mathbb{N}$.
2. For each $m \in \mathbb{N}$ there exists a least integer c_m such that $\pi^{c_m}(m) = \pi^{c_m+1}(m)$.
3. $\pi(m) = m$ iff $m = 24 \cdot 5^k$ for $k \in \mathbb{N}_0$ or $m = 1$.

4. If $m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$, then $\pi(m) = LCM(\pi(p_1^{\alpha_1}), \pi(p_2^{\alpha_2}), \dots, \pi(p_k^{\alpha_k}))$, where LCM denotes least common multiple and $p_1 < p_2 < \dots < p_k$ are primes.
5. **Conjecture:** if p is a prime number and $k \in \mathbb{N}$ then $\pi(p^k) = \pi(p) \cdot p^{k-1}$. This has been verified for all primes less than 10^{14} (see [8]).
6. $\pi(2) = 3, \pi(3) = 8, \pi(5) = 20$.
7. If $p > 5$ is a prime such that $p \equiv \pm 1 \pmod{5}$ then $\pi(p) \mid (p - 1)$.
8. If $p > 5$ is a prime such that $p \equiv \pm 2 \pmod{5}$ then $\pi(p) \mid 2(p + 1)$ and $\pi(p) \nmid (p + 1)$.

In order to develop a good algorithm, we will “believe” that the conjecture from Fact 5 is true, i.e. we will consider it as a valid theorem. There are two reasons for this: first, it is verified for all primes $p < 10^{14}$ and, since we deal with numbers less than 10^{18} , one can show that the probability of choosing a number from $[1, 10^{18}]$ with prime divisor $> 10^{14}$ is less than $\sum_{i=1}^x \frac{1}{p_i}$, where p_1, p_2, \dots, p_x are all primes from $[10^{14}, 10^{18}]$ (exercise for the reader: estimate the value of this sum). Second, if we hit a counterexample, we will get WA but we will disprove the conjecture and be famous ☺.

Now, combining Facts 4-8 we can get a straightforward algorithm for computing $\pi(m)$. Notice that we will need to factorize some numbers (Fact 4 and 7-8). Sieve of Eratosthenes and $O(\sqrt{m})$ factorization is slow as $m \leq 10^{18}$; some randomized algorithms are needed. We used **Pollard’s rho algorithm** in combination with **Miller-Rabin primality testing**. After factorization and using Facts 4-5, we have

$$\pi(m) = \pi(p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}) = p_1^{\alpha_1-1} p_2^{\alpha_2-1} \dots p_k^{\alpha_k-1} \cdot LCM(\pi(p_1), \pi(p_2), \dots, \pi(p_k)).$$

By the Facts 7-8, for prime number p , we only need to consider divisors of $p - 1$ or $2(p + 1)$ as candidates for $\pi(p)$ (there are $O(\sqrt{p})$ of them). Divisor d is a period of the sequence $\{F(n) \bmod p\}_{n \in \mathbb{N}_0}$ iff $F(d) \bmod p = 0$ and $F(d + 1) \bmod p = 1$ which can be checked with simple Fibonacci number calculation. Since $\pi(p)$ is the least period, we need the least divisor which is a period. More details and complexity analysis can be found in implementation section.

Fact 1 tells us that $\pi(m) = O(m)$ and it really fits into 64 bits and Fact 2 confirms our assumption on existence of c_m . If $p > 5$ is a prime, since $\pi(p) \mid (p - 1)$ or $\pi(p) \mid 2(p + 1)$ it follows that $\pi(p) < p$ unless $p \equiv \pm 2 \pmod{5}$ and $\pi(p) = 2(p + 1)$ which happens (approximately) in at most 50% of the cases for randomly chosen p . Also, from mentioned divisibility we have $\gcd(\pi(p), p) = 1$ and $\pi(p)$ has no prime divisor $> p$ (otherwise, that divisor has to be $\geq p + 2$, but $\pi(p) \leq 2(p + 1)$). It follows that the exponent of p_k in $\pi(m)$ decreases by 1, and it usually holds for many other exponents (some new primes may appear in $\pi(m)$). Therefore, it is expected that all the numbers $\pi^i(m)$ are of the same or less order of magnitude as m (and fit into 64 bits) and, since exponent-decrease argument and (usually) small values of k and α_i , that c_m is small. This was confirmed in practice – in almost all of the 1000 test cases $c_m < 20$. Now, Fact 3 says that we must have $\pi^{c_m}(m) = M = 24 \cdot 5^k$ for some $k \in \mathbb{N}_0$ and it is enough to analyze the numbers of that form for the estimation of the maximum cycle length C of the iterative process $a_i = F(a_{i-1}) \bmod M$. Result: even for moderately large values of k it often holds $C < 10^6$.

Conclusion: values c_m and C are small enough and calculation of $\pi(m)$ is fast enough – with some optimizations this algorithm passes randomized test cases within given time limit.

Implementation:

Out of 6 competitors who solved this problem, 5 of them (including the author of this text) had running time $\geq 37s$ (out of allowed 40s). This indicates that careful implementation and couple of optimizations are needed. Here we list some of them:

64-bit multiplication: During the algorithm, we often need to calculate expressions of the form $(a \cdot b) \bmod c$, where a, b and c are 64-bit integers. This is a problem – with simple multiplication the intermediate result will overflow. Type double is not precise enough and big number implementation (using arrays) is slow. Assuming that $2c$ fits into 64 bits (which is true since all numbers are $\leq 10^{18}$) the following (fast) recursive function $mul(a, b, c)$ which returns $mul(a, b - 1, c) + a$, if b is odd and $2 \cdot mul(a, b/2, c)$, if b is even solves the problem. Also, addition in the form “ $x = a + b$; if $(x > m)$ then $x = x - m$,” is much faster than “ $x = (a + b) \bmod m$,”.

Factorization: For a given composite n , Pollard’s rho algorithm returns (some) factorization $n = ab$ in asymptotically expected time $O(\sqrt{p})$ where p is the smallest prime factor of n (this is just a heuristics claim; rigorous analysis of the algorithm remains open). With first 11 prime numbers as “witnesses” in the Miller-Rabin algorithm, primality test can be done in $O(\log n)$ with 100% success rate for $n \leq 10^{18}$. Therefore, for $m = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$, factorization complexity is roughly $O(\sum_{i=1}^k \sqrt{p_i} + k \log m) = O(k(\sqrt[4]{m} + \log m))$. The last equation is due to the fact that if $p_i > \sqrt{m}$ then $\alpha_i = 1$ and we will never call Pollard’s rho for $p_i^{\alpha_i}$ – Miller-Rabin will “catch him” (there can be at most one such prime). For the calculation of $\pi(m)$ we also need to check the divisors of the numbers $x_i = p_i - 1$ or $2(p_i + 1)$. However, notice that we don’t need to check **all** divisors: if $x = q_1^{\beta_1} q_2^{\beta_2} \dots q_l^{\beta_l}$, we check the numbers $x/q_1, x/q_1^2, \dots, x/q_1^{\beta_1}$ and let $i =$ the largest index for which x/q_1^i is a period (possibly $i = 0$). Then we continue with q_2 with a new $x = x/q_1^i$ and so on; the final value of x is $\pi(x)$ (Why?). This way, we examine $\sum_{i=1}^l \beta_i = O(\log x) = O(\log p)$ divisors of x with $O(\log x)$ check complexity which is much better than $O(\sqrt{p})$ divisors. Therefore, the approximate complexity of calculating $\pi(m)$ is

$$O\left(k(\sqrt[4]{m} + \log m) + \sum_{i=1}^k l_i(\sqrt[4]{p_i} + \log p_i) + \sum_{i=1}^k \log^2 p_i\right)$$

where the first term corresponds to the factorization of m , the second term to the factorizations of $x_i = O(p_i)$ where l_i is its number of prime divisors, and the last term is the period search.

Period memoization: Since we are calculating values $\pi(m), \pi^2(m), \dots, \pi^{c_m}(m)$ and they are mostly decreasing, it is expected that many of them will share the prime factors. In order to avoid redundant calculations of Pisano period of those primes, we can use a **memoization technique** – a global hashmap in which we memorize $\pi(p)$ for all primes p encountered so far. This is further effective since all (1000) test cases will use this hashmap which will reduce the calculations.

Notes:

Since this problem requires implementation of many subroutines, it is recommended to start with the problems with codes PISANO, FRS2 and FRST from the SPOJ online judge; they can be used for extra testing of factorization, Pisano period calculation, cycle detection etc.

Some readers may have wondered why the “magic” number 5 appears in the Pisano period Facts 7-8. The answer is not obvious and lays in the well-known Binet’s formula:

$$F(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right).$$

The elements $\phi = \frac{1+\sqrt{5}}{2}$ and $\phi^- = \frac{1-\sqrt{5}}{2}$ play significant role and it would be nice if they were an integers –

we could use Fermat's little theorem to find periods. It can be shown that 5 is quadratic residue modulo prime number p (i.e. there exists integer x such that $x^2 \equiv 5 \pmod{p}$) iff $p \equiv \pm 1 \pmod{5}$. If 5 is quadratic residue modulo p then we can imagine that $\sqrt{5}$ is an integer in the field Z_p (the corresponding x from the congruence relation) and so are ϕ and ϕ^{-1} and $1/\sqrt{5}$ (as multiplicative inverse of x); otherwise, we can analyze them as the elements of the quadratic extension field $Z_p[\sqrt{5}]$ (i.e. field Z_{p^2}). Further (and non-trivial) analysis using Lagrange's theorem regarding group order yields Facts 7-8.

Solution by:

Name: Nikola Milosavljević

School: Faculty of Sciences and Mathematics

E-mail: nikola5000@gmail.com

Problem R2 06: Help Blue Mary Please! (Act I) (ID: 1457)

Time Limit: 17.0 second

Memory Limit: 256 MB

This morning Blue Mary wrote some equations on a piece of paper and left it on her desk. After solving some problems in SPOJ, she found that her classmate H.L. replaced all characters on the paper with some other ones. H.L. told her he replaced the same characters with the same ones, and different characters with different ones because of his goodness. Now Mary needs your help to get the original equations back.

In Mary's equations, only 13 characters appear: 0,1,2,3,4,5,6,7,8,9,+,*,=. There is one and only one "=" in each equation. In H.L.'s equations, only 13 Latin letters appear: a,b,c,d,e,f,g,h,i,j,k,l,m. All the equations are correct in decimal notation.

For example. If Mary wrote down $2+29=31$, H.L. replaced 2 with i, + with l, 9 with k, = with e, 3 with m and 1 with a, we got $ilikema$.

Input

The first line contains a single integer t . t blocks follow.

To every block, the first line contains a single integer n ($1 \leq n \leq 1000$). n lines follow, each contains a string whose length is more than 4 and less than 12. The string contains only a-m and doesn't contain any whitespaces.

At least 90% of test cases satisfy that $n \leq 5$.

At least 80% of test cases satisfy that $n \leq 2$.

In at least 70% of test cases, there are at most 5 different characters in all the equations.

Output

If there doesn't exist n equations that can be translated to H.L.'s equations, print a line contains the word *noway*. Otherwise you should output all the corresponding relations that can be fixed in lexicographic order, see the example.

Sample

input	output
1	a6
2	b*
abcdec	d=
cdefe	f+

Hint

The two strings can be translated to the following equations possibly:

$$6 * 2 = 12 \quad 2 = 1 + 1$$

$$6 * 4 = 24 \quad 4 = 2 + 2$$

$$6 * 8 = 48 \quad 8 = 4 + 4$$

So the corresponding relations above can be fixed, others can not.

Solution:

You are asked to output all possible combinations that satisfy given equations, so we have to try all possibilities, therefore the backtracking approach seems like the right one. Since there are many

combinations, we have to do some optimizations. There are probably many approaches to solve this problem, but we are going to list a few that are good enough to satisfy the given time limit.

Since there is only one equals sign, we are going to set it first. A letter can be an equals sign only if it appears only once in every equation and it's neither the first nor last letter in that equation.

After the equals sign, we are going to set plus signs. This can be any letter that is not next to the letter that represents the equals sign, nor at the start or end of the entire equation. Also, we can say that there is no plus sign and to move on.

Next comes the multiplication sign, which has the same rules as a plus sign, only we have to add rule that multiplication sign can't stand next to the plus sign. Same as with plus sign, we can say that there is no multiplication signs and we can move on to the next step.

Once we have set all signs, only digits are left. Significant optimizations are needed here, since there are many combinations left, because there are 10 digits for every remaining letter.

We have n equations, and after setting a new letter as a digit, we will go through all equation and check if there is a point to move on, or we can say in that moment that current pairing of letter with signs and digits is not possible. For every equation we are going to consider left and right part of equal sign.

We will calculate, as accurately as possible, the minimal and maximal number we could get with current settings on both sides of the equals sign. Let's call those numbers are *minLeft*, *maxLeft*, *minRight* and *maxRight*. It's clear that if $minLeft > maxRight$ or $maxLeft < minRight$ there is no point to move on with the current set of letter transformations, and we can break here.

Now we are going to see how to calculate the minimal (maximal) number on one side of equal sign. Since we have already set signs, we can split expression in buckets where each bucket would represent one number and there is a sign (plus, multiplication) between two consecutive buckets.

In order to minimize (maximize) the expression it's enough to minimize (maximize) each bucket. Each bucket is made of digits, letters that are paired with some digit, and letters that are still not paired. So we just need to pair those letters with digits in such a way that the current bucket is minimal (maximal). We can do that if we follow the rule that the letter that comes first has to be smaller (bigger) than the letter that comes after, not taking in consideration letters that are already paired with some digits.

This last optimization would lead to a fast enough solution to pass all test cases. The most problematic part of this problem is coding, since this has to be coded in an efficient way. The coder has to follow the rule of backtracking and not do anything twice, as well as not do anything stupid or insufficient.

Solution by:

Name: **Demjan Grubić**

School: PMF

E-mail: demjangrubic@gmail.com

Problem R2 07: Problems Collection (Volume X) (ID: 1815)

These ten problems come from Chinese National Olympiad in Mathematics - Province Contest.

Problem 1 Polynomial $P(x) = x^5 + a_1x^4 + a_2x^3 + a_3x^2 + a_4x + a_5$, and we know when $k = 1, 2, 3, 4, P(k) = 2007 * k$. Calculate $P(10) - P(-5)$.

Problem 2 The sum of 100 positive integers a_1, a_2, \dots, a_{100} is 2007. Calculate the maximum possible value

of $\sum_{1 \leq i < j < k \leq 100} a_i a_j a_k$.

Problem 3 Calculate $100101102103104 \dots 498499500$ modulo 126.

Problem 4 We define the sum of the first n numbers of geometric progression $\{a_n\} S_n$. Now we know $S_7 = 7, S_{14} = 2014$. Calculate $S_7 * (S_{21} - S_{14})$.

Problem 5 Calculate the sum of this kind of positive integers $n (n \geq 4)$: n satisfies that $n!$ can be written as the product of $n - 3$ consecutive positive integers.

Problem 6 Two vertices of a square are on the line $y = 2x - 17$, while the other two are on the parabola $y = x^2$. Calculate the sum of two different possible values of the area of this square.

Problem 7 A, B, C, D are four fixed points in the space and they are not on the same plane. Calculate the number of different parallelepipeds, which satisfies that 4 vertices of the parallelepiped are A, B, C and D .

Problem 8 Polynomial $x^2 - x - 1$ exactly divides Polynomial $a_1x^{17} + a_2x^{16} + 1$. Calculate $a_1 * a_2$.

Problem 9 Suppose x is an acute angle, calculate the minimum possible value of $(\sin x + \cos x)/(\sin x + \tan x) + (\tan x + \cot x)/(\cos x + \tan x) + (\sin x + \cos x)/(\cos x + \cot x) + (\tan x + \cot x)/(\sin x + \cot x)$.

Problem 10 Suppose $x^4 + y^4 + z^4 = m/n$, x, y, z are all real numbers, satisfying $x * y + y * z + z * x = 1$ and $5 * (x + 1/x) = 12 * (y + 1/y) = 13 * (z + 1/z)$; m, n are positive integers and their greatest common divisor is 1. Calculate $m + n$.

Input

There is no input.

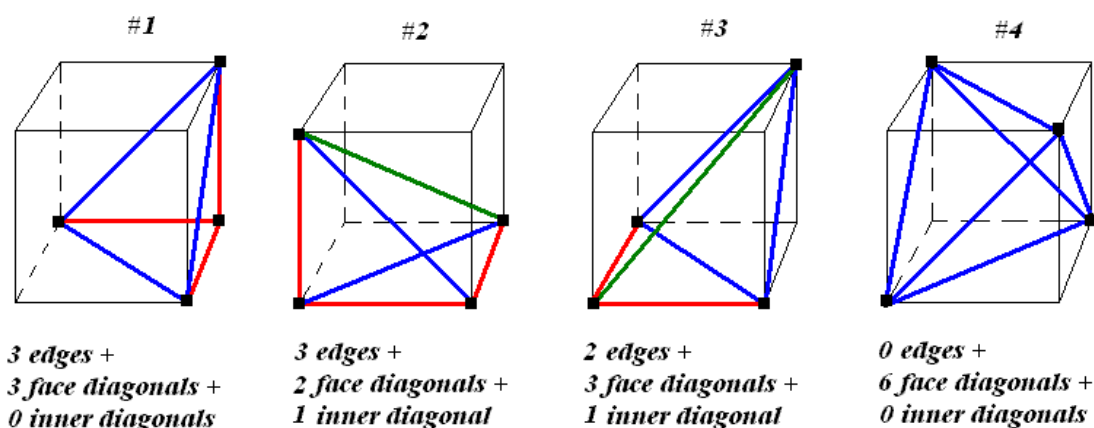
Output

Ten lines, each contains a single integer denoted the answer to the corresponding problem.

Solution:

1. This task is rather simple and straightforward. One of possible solutions is to use successive differences. By doing that we can notice a_5 is irrelevant so we can reduce a system with 5 variables and 4 equations to a system of 4 variables and 4 equations. From here it is simple math, but if you get stuck you can use a tool on your PC like Microsoft Mathematics, or an online tool like Wolfram Alpha.
2. Consider two terms x and y among the a -values, where $x < y - 1$. We can easily notice that by decreasing y by one and increasing x by the same amount, the sum of two numbers stays the same, but value of their product increases. Knowing this we can write a small piece of code to handle the calculation of the sum.
3. There are many different ways of solving this one. The simplest is to use a library for working with big integers and just performing the module operation. Another one would be breaking this number into sums of smaller numbers, performing mod operations on them separately, and then calculating the sum using modulo arithmetic.

4. If we use the formula for the sum of terms of a geometric sequence $S_n = a(1 - r^n)/(1 - r)$ we get two equations with two variables. It is straightforward from here, but you can use some tricks (might be better not to calculate a_1 and r directly) to avoid messy calculations or you can use math help tool.
5. We should find the maximum value of n first. Let the largest of the $n - 3$ consecutive positive integers be k . Clearly k cannot be less than or equal to n , or the product would be less than $n!$. Let us try to find the largest value of n for a constant value of $k - n$. If $k = n + 1$, the $n - 3$ consecutive positive integers are 5, 6, 7 ..., $n + 1$. So we have $(n + 1)! = \frac{n!}{4!} \Rightarrow n + 1 = 24 \Rightarrow n = 23$. If $k = n + 2$, we have to add a 5 on one side of the equation and $n + 2$ on the other, which means that the maximal value of n will be smaller than in the case when $k = n + 1$. Since the same argument still applies when we increase $k - n$, our overall maximal n is 23. Finding smaller values of n is straightforward ($n \in \{7, 6, 4\}$).
6. We need to compute four vertices, so a lot of variables to solve for but also there is lot of data, so some messy algebra could be required. We should set up a system of equations by using the fact that vertices are on given lines, then we should use the fact that cross-line (line connecting vertices on different curves representing one of the square edges) is perpendicular ($k_1 \cdot k_2 = -1$, where k_1 and k_2 are line coefficients) to the straight line ($y_2 = 2x_2 - 17$) and that the distance between vertices is equal (we can calculate edge length by multiplying line offsets with $\tan \alpha = k_2 = 2$). Once we have that set I recommend using an equation solver.
7. How can we inscribe a tetrahedron in a parallelepiped (i.e. to choose 4 non-coplanar out of 8 of parallelepiped's vertexes as tetrahedron's vertexes)? The inscribed tetrahedron will be one of the following 4 types:



Its 6 edges will be:

- 3 edges (red) meeting in a vertex, plus 3 face diagonals (blue) of the initial parallelepiped;
- 3 edges like a twisted 'Π', plus 2 face diagonals, plus 1 inner diagonal (green);
- 2 edges and inner diagonal with common vertex, plus 3 face diagonals, connecting their other endpoints;
- 6 face diagonals (one per face).

Calculating the sum of parallelepipeds for each case is solution to the problem.

8. We can apply the quadratic formula and the Factor Theorem directly but a nicer solution would be

finding a pattern as we solve smaller cases (for $a_1 x^n + a_2 x^{n-1} + 1$, investigating $n = 2, 3, 4, \dots, 17$). Once found, we end up with an easily solvable system of two equations.

9. Calculating the derivative of the expression, then finding the value of x when the derivative is zero and then putting it back in our original expression (with help of some tool like Microsoft Mathematics) would save us a lot of trouble and would produce the correct solution.
10. It would be best to start by solving the system of equations without variables m and n . Should be straightforward (three equations with three variables) but could get messy so a math tool is recommended. If everything is correct we should have five different solutions. Now we test them for satisfying $x^4 + y^4 + z^4 = m/n$. Three are eliminated on first sight so only two are left for testing, and one of them fits perfectly.

Solution by:

Name: **Nikola Obradović**

School: Faculty of Computing

E-mail: nobradovic09@gmail.com

Problem R2 08: AB-words (ID: 177)

Time Limit: 13.0 second

Memory Limit: 256 MB

Every sequence of small letters a and b (also the empty sequence) is called an ab-word. If $X = [x_1, \dots, x_n]$ is an ab-word and i, j are integers such that $1 \leq i \leq j \leq n$ then $X[i..j]$ denotes the subword of X consisting of the letters x_i, \dots, x_j . We say that an ab-word $X = [x_1..x_n]$ is nice if it has as many letters a as b and for all $i = 1, \dots, n$ the subword $X[1..i]$ has at least as many letters a as b .

Now, we give the inductive definition of the similarity between nice ab-words.

- Every two empty ab-words (i.e. words with no letters) are similar
- Two non-empty nice ab-words $X = [x_1, \dots, x_n]$ and $Y = [y_1, \dots, y_m]$ are similar if they have the same length ($n = m$) and one of the following conditions is fulfilled:
 1. $x_1 = y_1, x_n = y_n$ and $X[2..n-1]$ and $Y[2..n-1]$ are similar ab-words and they are both nice;
 2. there exists $i, 1 \leq i \leq n$, such that $X[1..i], X[i+1..n]$ are nice ab-words and
 - a. $Y[1..i], Y[i+1..n]$ are nice ab-words and $X[1..i]$ is similar to $Y[1..i]$ and $X[i+1..n]$ is similar to $Y[i+1..n]$, or
 - b. $Y[1..n-i], Y[n-i+1..n]$ are nice ab-words and $X[1..i]$ is similar to $Y[n-i+1..n]$ and $X[i+1..n]$ is similar to $Y[1..n-i]$.

A **level of diversity** of a non-empty set S of nice ab-words is the maximal number of ab-words that can be chosen from S in such a way that for each pair w_1, w_2 of chosen words, w_1 is not similar to w_2 .

Write a program that for each test case:

- reads elements of S from standard input;
- computes the level of diversity of the set S ;
- writes the result to standard output.

Input

The number of test cases t is in the first line of input, then t test cases follow separated by an empty line.

In the first line of a test case there is a number n of elements of the set $S, 1 \leq n \leq 1000$; in the following n lines there are elements of the set S , i.e. nice ab-words (one word in each line); the first letter of every ab-word is the first symbol in line and there are no spaces between two consecutive letters in the word; the length of every ab-word is an integer from the range $[1..200]$.

Output

For each test case your program should output one line with one integer - the level of diversity of S .

Sample

input	output
1 3 aabaabbbab abababaabb abaaabbabb	2

Solution:

First of all, let's transform this problem into a graph theory problem. Each word represents a vertex. Two vertices are connected if and only if the words corresponding two vertices are similar. Then the solution to the problem is to find the maximum independent set in this graph. Maximum independent set is the largest set of vertices in a graph, no two of which are adjacent. This is an NP problem, and so far, there hasn't been

found a solution that runs in polynomial time. The solutions to this problem have exponential complexity and without any optimizations, they are not good enough.

If we replace every letter 'a' with '(' and every letter 'b' with ')', the word is nice if the resulting sequence of parenthesis is well formed. This expression can be represented by a forest, as shown in figure 1. To reduce it to a rooted tree, we will add an 'a' at the beginning of the word and 'b' at the end of the word.

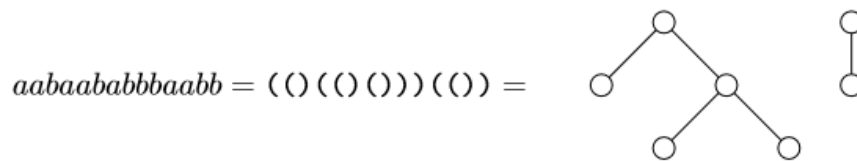


Figure 1. Representation of AB-word as a parenthesis expression and as a set of trees

It is not hard to see that if the trees corresponding to two words are not isomorphic, then the words are not similar. This can be used as an optimization. We will make classes of pairwise isomorphic trees and test for similarity only between words in the same class. Sum of the sizes of maximum independent sets in each class is the solution.

Two trees are isomorphic if and only if they have the same canonical form. One way to transform the tree into a canonical form is using the following algorithm (source: http://webhome.cs.uvic.ca/~wendym/courses/522/07/1notes/tree_can.html):

1. Label each vertex with the 2-character string (). Set the phase=0.
2. While more than two vertices remain do
 1. Locate all the leaves.
 2. Remove each leaf placing its label within the (and) of the parent. Do this so that within the () of the parent, the strings representing the children are sorted, shortest labels first, then use lexicographic order. The criteria selected for sorting is not important as long as it is consistent and the rules do not depend on any labelling that has been applied to the tree, only to its isomorphism class. Another alternative is to use just lexicographic order.
 3. Increment phase.
3. If there is one vertex left, the canonical form of the tree is the label of this vertex. If there are two vertices u and v left, the canonical form of T is $label(u)$ concatenated with $label(v)$ where $label(u) \leq label(v)$.

Since the trees are already given in the form of words (or equivalently the parenthesis expression), we do not have to actually build the trees and instead we can just use the string representation. We can then sort the resulting strings and group them into isomorphism classes.

As said, we now have to test for similarity only between words of the same class. Testing for similarity can be done recursively, according to the definition of similarity of AB-words in the task. With careful implementation and some small optimizations in the backtracking algorithm for finding maximum independent sets of each class, this is enough to pass all the test cases. Detailed explanation of the previous ideas is given here: <http://www.oi.edu.pl/static/attachment/20110731/oi5.pdf> (in Polish)

Solution by:

Name: **Vanja Petrović Tanković**

School: Faculty of Computing

E-mail: vpetrovictankovic@gmail.com

Problem R2 09: Santa Claus and the Presents (ID: 240)

Time Limit: 17.0 second

Memory Limit: 256 MB

Every year Santa Claus faces a more and more difficult task. The number of children in the world is increasing rapidly, while Santa's old patched up sack can only accommodate a few presents at a time. And every child wants their own very special present... This means that, ever so often, once his sack is partially or completely empty, Santa has to fly back to his base in Lapland to replenish his supplies. So irksome has this become that Santa has decided to resort to modern operational research to aid him in his sack-packing and route planning. Please write a program which will guide Santa through his daily chores.

Input

The input starts with a line containing a single integer $t \leq 100$, the number of test cases. t test cases follow.

The first line of every test case consists of four integers $n \ x \ y \ S$, denoting the number of good children who will receive a present from Santa, the x and y coordinates of Santa's home base, and the amount of space in the sack, respectively ($1 \leq n \leq 10000$, $-10000 \leq x, y \leq 10000$, $1 \leq S \leq 100000$). n lines follow, each consisting of three integers $x_i \ y_i \ s_i$ - the x and y coordinates of the i -th child's home, and the amount of space taken up by this child's present when in Santa's sack, respectively ($-10000 \leq x_i, y_i \leq 10000$, $1 \leq s_i \leq S$).

Output

For each test case output a sequence of space separated integers, corresponding to successive actions that should be taken by Santa:

$-i$ ($1 \leq i \leq n$) signifies that Santa should travel to his base and pack the present for the i -th child into his sack (he needs to have sufficient room in his sack to do this).

i ($1 \leq i \leq n$) signifies that Santa should travel to the i -th child's home and leave a present for him/her.

0 signifies that Santa should travel back to his base and end his Christmas activity (and that you want to proceed to the next test case).

Assume that Santa starts in his base and always travels between points by the shortest possible route (along straight lines). All distances are measured using the Euclidean metric.

Sample

input	output
1 3 0 0 3 1 0 1 1 0 2 1 0 3	-1 -2 1 2 -3 3 0

Score:

$$2/(1+1+1+1)=0.5$$

Score

The score awarded to your program is the total of all scores obtained for its individual test cases. The score for a test case is calculated as I/P , where P stands for the distance covered by Santa in your solution, while I is a test case specific constant ($I = n * d + D * (s_1 + \dots + s_n)/S$; d is the mean distance between two homes, while D is the mean distance between Santa's base and a child's home for the given test case).

No points are awarded for incompletely solved test cases (when not all the children receive presents). Any other violation of transport rules results in Wrong Answer.

Solution:

This is a variation of the well-known Traveling Salesman Problem. It's known that it is an NP-hard problem. Though it is NP-hard, and it requires exponential time complexity in order to get the best result, there are many heuristics that work faster and give "good" results.

Here's a list with just a few of them:

- Nearest Neighbor $O(n^2)$
- Nearest Insertion $O(n^2)$
- Cheapest Insertion $O(n^2 \log n)$
- Stewart Algorithm $O(n^2 \log n)$
- Simulated Annealing

Let's try to combine these, to get as good result as we can.

First, we can use the Nearest Neighbor heuristic, which works as follows:

- 1) Start from an arbitrary point.
- 2) While there are unvisited points find the closest point to the last added point and connect to it.
- 3) Connect the last point to the first point.

Running time of this algorithm is $O(n^2)$, but we can speed it up using a KD-Tree.

We'll use KD-Tree here to store, organize and operate points representing children homes. Operations:

- Construct – Construction of a balanced 2D-Tree in $O(n \log n)$, n is the number of nodes in a tree.
- Nearest Neighbor Search – Finds the point in tree nearest to a given point in $O(\log n)$ average.
- Delete – Removes a point from 2D-Tree in $O(\log n)$ average.

As we might not be able to put all the presents in Santa's sack we may have several tours. Each time our tour weight, sum of the children spaces taken up by the presents, is more than Santa can handle he'll return home and start a new tour.

Steps overview:

- 1) Santa goes to home and starts a new tour.
- 2) If all houses are visited, Santa goes home.
- 3) If there's no more space in the sack, go to the step 1.
- 4) Santa finds the closest unvisited house and moves to it.
- 5) Go to step 2.

Alright! 385.854953 points in 12.98 seconds.

Ok, now let's try to improve each of the tours independently. If the size of some tour is small enough (for example up to 100 points – we can tune this over different test cases) we can apply Stewart heuristics and get a better result.

Here's layout of Stewart Algorithm:

- 1) Start with a convex hull as the initial tour.
- 2) If all points are visited, finish!
- 3) For each point k in unvisited points find its ratio: $(dist(i, k) + dist(k, j)) / dist(i, j)$ where i and j are two different visited points that minimize $cost(i, k, j)$ function.
- 4) Select point k which has the minimum ratio and insert it between points i and j on the tour.

5) Go to step 2.

$dist(a, b)$ = the Euclidean distance between points a and b .

$cost(a, b, c) = dist(a, b) + dist(b, c) - dist(a, c)$.

389.883505 points in 15.97 seconds. Not bad but not the best.

We get one more improvement by iterating through all points on the individual tour:

- 1) Take the tour obtained by Stewart Algorithm, now current point is the first one (Santa's home).
- 2) Take the current and next 6 points on it, if there are no 6 points left – end.
- 3) Rearrange those 6 points in a way that minimizes distance from the current to the sixth point.
- 4) Move to the first next point and go to the step 2.

393.152092 points in 16.97 seconds.

Solution by:

Name: Dimitrije Dimić

School: Faculty of Computing

E-mail: dimke92@gmail.com

Problem R2 10: Robo Eye (ID: 2629)

Time Limit: 5.0 second

Memory Limit: 256 MB

A robot which helps to translate old papers into digital format is being prepared for mass production. But it requires special software to work efficiently. All scans which will be analyzed by the robot use one of the standard fonts: Arial, Courier, or Times New Roman. As the press is not ideal and pages can be rotated, some of the scans will be rotated by some degree. Imperfections of the robot's camera ("eye") can also cause some noise. Uniform noise can be up to 2% of the number of pixels. All analyzed documents are in uppercase English. The total number of letters in the robot's eye ranges from 3 to 6. The matrix in the robot's eye is monochrome and its size is 200x200 pixels.

Input

t – the number of tests [$t \leq 500$], then t tests follows. Each test consists of 200 rows with 200 chars in each of them. Characters can be '.' and 'X', where '.' means the white color of the page, and 'X' is the black color of words.

The input data was generated using an on-line generator. The generator outputs data in 2 formats:

1) [as a picture](#)

2) [as text](#)

The datasets used for testing are such that all letters of the word are contained inside the picture.

Output

For each test output the recognized word in a separate line.

Score

The number of points you'll receive for each image will be equal to the number of letters of the word, provided that it is correctly recognized.

Solution:

Problem Robo Eye on this year's Bubble Cup qualifications was a challenge problem which required Optical character recognition (OCR). Given a binary image of size 200x200 containing a word of length 3 to 6, with possible rotation and noise, output the word.

There are many attempts to solve this problem and one of them is doing the following:

1. Remove noise
2. Locate text
3. Rotation deskew
4. Recognize text
5. Put recognized text through a text filter
6. Output recognized text

The first step of the algorithm is to remove the noise. It is done easily by finding and discarding all connected components of size less than some threshold (in my case it was 20).

The second step is to locate the text, find the bounding box of the text in the image. And since all the noise is removed, all the content left in the image is text content. In order to find the bounding box the most top, right, bottom and left pixels have to be found to locate the position of the text.

The third step is to rotate the image so that the text is not rotated (it is horizontal). There are various algorithms to do that. The simplest one is to rotate the image for each degree between -90 and 90 and pick one for which bounding box of the text has the lowest height (highest width). This will work because the word in the image contains at least three letters, so its width will be larger than its height. So if the word is rotated by some angle counter-clockwise, rotating it to the right will decrease its bounding box height and increase width, and rotating it to the left will increase its bounding box height and decrease width. It is analogous for the word rotated for some angle clockwise.

It is important to notice that if for some two angles X and $X - 1$ the bounding box produced by angle X has lower height than the bounding box produced by angle $X - 1$, all angles Y less than X will produce the bounding box with higher height than bounding box produced by angle X . This allows us to use binary search and find the appropriate angle. It is required to check $(\log 180)$ angles, which is less than 10, until the right one is found. Finding the bounding box height requires going through each pixel of the image again, which has $O(\text{width} \cdot \text{height})$ complexity, so the complexity of deskewing the image is $O(\log 180 \cdot \text{width} \cdot \text{height})$. A good optimization is to iterate only through pixels that belong to the rotated bounding box, as the bounding box is usually a lot smaller than the whole image. Another good approach is to calculate the position after rotation only for black pixels (text pixels), and then the complexity would be $O(\text{number of black pixels})$, which is much faster, but it wasn't necessary for solving the problem.

Note: There could be a few angles (up to ± 2 degrees) which give the lowest bounding box. In that case any angle could be taken since the skew is small and will not affect text recognition much in this task.

One more approach that is worth mentioning is finding the line P which underlines the text and then rotate the image so that the line P becomes horizontal. Finding the line P can be done by using Hough Transform.

Let's say we want to rotate point (x_0, y_0) around point $(0, 0)$ by some angle φ . Using simple trigonometry we get following equations:

- $x = x_0 \cdot \cos \varphi - y_0 \cdot \sin \varphi$
- $y = x_0 \cdot \sin \varphi + y_0 \cdot \cos \varphi$

The problem here is that (x, y) coordinates are real numbers and what is the best corresponding pixel in that case. The simplest interpolation method, which is good enough for solving the task, is remove fraction from point (x, y) and count the black pixels in the 3×3 area around it. If number of black pixels exceeds some threshold (half of 3×3 area is good, which is 4) then point (x_0, y_0) is black, otherwise it is white.

The fourth step is recognizing the horizontal text. It can be split in two steps. First one is separating letters and the second one is performing OCR for each letter. Before explaining step one, it is required to understand how step two works.

OCR algorithm:

Suppose function $f(\text{image})$ returns some value of the image. For each letter of each font an image with that letter can be created and its value can be calculated using the function $f(\text{image})$. So there is a mapping for each letter of each font, $(\text{font}, \text{letter}) = f(\text{image}(\text{font}, \text{letter}))$, where $\text{image}(\text{font}, \text{letter})$ is an image containing letter of the corresponding font. The idea is to find function f such that the closer the difference of the values is, the better letter match is found.

Separating the letters:

Usually it is enough to find connected components and suppose that each component is one letter. But it is not always the case due to imperfect rotation, so some letters might be joined. Good assumption is that if width of the connected component is less than some threshold (the best value can be found from real sample tests, and it is around ten) it is one letter for sure. Otherwise, some vertical line X can be fixed and suppose that the content of the connected component from the beginning to the vertical line X is one letter, which is the first part of the image. Calculate the value $K = f(\text{the first part of the image})$. The split line is line X for which value K is minimal (the best match is found). Process is repeated until the width of the remaining content is higher than double width of the threshold (width of the letter).

Finding the function f :

The main property of function f should satisfy that, the closer the values are, the higher letter match it is, and viceversa. So we have to find some features of the image that satisfy these conditions. Let's define the value of function f as vector $V(a_1, \dots, a_n)$. Let the difference between two vectors $V_1(a_1, \dots, a_n)$ and $V_2(b_1, \dots, b_n)$ be $\sum_{i=1}^n c_i * (a_i - b_i)$ where c_i is some corresponding coefficient.

Features of function f :

One of the best features is dividing the image to a $N \times M$ grid and for each field of grid calculating the average number of black pixels. So the first $N \times M$ values of f would be these average values.

The second feature is ratio of width and height of letter.

These features are good but can't always tell the difference between some letters, for example H and N , F and P , O and Q , R and B , etc. Of course, it wouldn't be a problem if the image was of higher resolution, but in this task it is low so we have to come up with more features.

The third group of features is helping to make the difference between O and U , B and R , ... We are adding the distance from each side (top, right, bottom and left) until the first black pixel is reached. (image 1). Note: for letters O and Q distance from the bottom-right corner can be added as a feature if necessary.

Image 1. Yellow, red, green and blue lines are showing the distances from top, right, bottom and left sides



respectively

Fourth feature is number of connected components of white pixels which are not touching the bounds. For example, letter P would have one and letter F would have zero. Letter R would have one and letter B would have two. It helps a lot when there is a confusion between F and P .

Fifth feature is helping us to make the difference between H and N . It counts the number of intersections of middle horizontal line. For example, H has one intersection and N has three intersections. Note that here intersection is defined as number of changes from white to black pixel and black to white.

We still have to define coefficients (c_1, \dots, c_n) . At the beginning all coefficients can be one and after a lot of tests are acquired then these coefficients can be changed in order to achieve better accuracy.

Finally, for each font and each letter values of function f should be calculated and stored in the source code, so the values can be used for finding the best letter match.

After text is recognized it is good to put it through some check. Sometimes there might be two letters V next to each other, where it is hard to say whether it is a double V or a W . So if there are two letters V next to each other and the word's length is 7 then it is for sure W and not double V because maximal word length is 6. If the word's length is less than 3 and the W is recognized then it is for sure double V . In other cases it can't be determined for sure, but it can be checked which solution gives a better score and take that one.

Time complexity of the solution is: $O(\text{width} \cdot \text{height} + \text{numberOfLetters} \cdot \text{numberOfFonts} \cdot \text{numberOfFeatures})$, but the constant here is very high, depending on kinds of features that are used, because calculating feature might be $O(1)$, $O(\text{height})$, $O(\text{width})$ or $O(\text{width} \cdot \text{height})$.

Memory complexity of the solution is: $O(\text{width} \cdot \text{height} + \text{numberOfLetters} \cdot \text{numberOfFonts} \cdot \text{numberOfFeatures})$.

Solution by:

Name: Nikola Stojiljković

School: Faculty of Computing

E-mail: nikolavla@gmail.com

*The scientific committee would like to thank everyone
who did important behind-the-scenes work.
We couldn't have done it without you!*

*We will prepare a lot of surprises
for you again next year!
Stay with us!*

Bubble Cup Crew





Microsoft®
Development Center Serbia



Government of the Republic of Serbia
Ministry of Education



UNIVERZITET U BEOGRADU
UNIVERSITY OF BELGRADE

bubble
cup



STUDENT CODING COMPETITION



INFORMACIONO DRUŠTVO SRBIJE
Društvo za informacione sisteme i računarske mreže

rc etf



računski centar elektrotehničkog fakulteta

