Topic 0: Introduction, Implementation **Lecture Notes**
CS 41100 - CP3 Competitive Programming III (Spring 2024) Purdue University
Instructor: Zhongtang Luo **Date:** January 16, 2024
Notes Preparer: Zhongtang Luo

1 Introduction

Everything here is also in the syllabus! Please also read the syllabus for more information.

1.1 Performance Task

Every Thursday, we will have in-class contests using three homework problems. Solving these problems in and out of class will give you credit towards this course. The composition may vary from week to week, but is usually

- A problem that checks prior knowledge on the topic of the next week (easier).
- A problem that tests the topic of this week.
- A problem that tests implementation skills.

You are not expected to solve everything in class, so you should read all the problems before deciding which one to solve. In general, we would expect an A student to solve only 1 problem in the contest, and upsolve the rest after the contest, while an A+ student should be able to solve 2 problems in the contest. So there is no need to panic if you find the problem a bit on the difficult side!

1.2 Student Topics

You will vote on two topics in this course. You may also present them for extra credit. Details are in the syllabus.

1.3 Lecture Notes

You may take and share lecture notes (just like this one) for extra credit. They should be written to a standard that every reasonable student in this class can understand. One lecture note is worth **6 pts**.

1.4 Concerns from Last Semester

The instructor assumes prior knowledge (e.g. network flow) that I do not know! Unfortunately, competitive programming at college level builds on so many different blocks (number theories, combinatorics, probability, etc. just to name a few from maths). It very quickly becomes difficult to progress to the more exciting parts without assuming that the students have general undergrad knowledge on relevant fields.

Nevertheless, everyone can use a little brush-up from time to time before we progress to the more difficult part of the course. Therefore, a review guide will be provided on the course page with practice problems. A review problem will also be included for the next week's topic every contest to help with reviewing (or learning) prior knowledge.

The slides are too concise and I don't know how to review! In general, college students have found note-taking to be helpful when reviewing. To promote the process, extra credit will be offered for taking and sharing lecture notes.

I spend 20 hours on homework problems every week! While this course is short (12 weeks) and thus has the same course load per week as a 3-credit course, we only expect around 6 hours every week on homework problems. If you are spending significantly more time, please come to the office hour so we can find a way to help you learn more efficiently.

With these logistics out of the way, let us move onto the course material.

2 General Introduction

In this section we will be looking at various problems we will tackle throughout the semester.

2.1 Basics

> **Sample Problem:** B Road Band
> **Link:** `https://vjudge.net/problem/Gym-104757B`

As you enter CP3, you already have command of a plethora of knowledge from CP1 and CP2. It is essential that you review it from time to time, since those problems are easier and thus more likely to appear in every contest. As a practice, try to solve this problem from ECNA 2023.

We will also try to promote the review process within contests. As a result, every contest has one easy problem from CP1/CP2 that is solvable in 30 minutes. It will be your first mission to identify the easy problem and solve it, as solving it in the contest also grants you a path to A in this course.

If you ever feel stuck on one of the easy problems, there is no need to panic. Everyone can use a bit of review from time to time, and you can feel free to drop in CP1/CP2 sessions to further your understanding of the material. You can also drop by my office hours, if you feel stuck at some point.

2.2 Meta in ICPC Contests

As we advance out of CP1/CP2, we will inevitably face advanced problems from ICPC contests. I personally categorize ICPC contests into three flavors.

2.2.1 Technique-Heavy (Chinese)

> **Sample Problem:** Cyclic Substrings
> **Link:** `https://vjudge.net/problem/Gym-104857C`

In a lot of Chinese ICPC regional rounds and Chinese national OI rounds, you will face problems solvable if and only if you know some advanced data structure technique. For instance, this problem is trivially solvable if you understand palindrome automata, and not solvable otherwise.

2.2.2 Observation-Heavy (Russian)

> **Sample Problem:** Mathematical Problem
> **Link:** `https://vjudge.net/problem/CodeForces-1916D`

In a lot of Russian rounds (Open Cup, Codeforces, etc.) you will face problems that rely heavily on observation and construction. For instance, this sample problem requires a clever construction.

2.2.3 Implementation-Heavy (US)

> **Sample Problem:** Forest for the Trees
> **Link:** `https://vjudge.net/problem/Gym-104757G`

In US rounds (particularly ECNA) you will face problems that are not particular hard to find a solution but difficult to implement, including computational geometry problems and pure implementation tasks. For instance, this sample problem from ECNA 2023 is not particularly hard to think about, but posed sufficient problems for the team here at Purdue so that we did not advance to NAC.

In this course, we aim to give a relatively balanced view on different problems. Nevertheless, since we are in the US, some emphasis will be put on implementation: one problem in each contest will be dedicated to implementation problems, which we will introduce today.

3 Implementation

> **Sample Problem:** ICPC Ranking
> **Link:** `https://vjudge.net/problem/HDU-4789`

Sometimes in contests, we are asked to implement straightforward tasks. While implementation seems a lot easier than thinking about algorithms, given the time constraint it is very important that we code as efficiently as possible. In fact, considering that we are holding contests in-class, you should aim to solve these kind of problems in under one hour!

General hints on implementation:

Think before you code. Try to figure out how many functions you will write and how many lines of code are there in each function. This forces you to mentally walk through the code so that you can grasp what you will be writing. Some people also find it helpful to prepare the structure of the code on a piece of paper before starting to implement.

Reduce code duplication. While it is tempting to copy and paste code around, it makes the code much less maintainable. Try to use techniques below to ensure maximum conciseness.

3.1 Recognition

> **Sample Problem:** Sumdoku
> **Link:** `https://vjudge.net/problem/Baekjoon-15303`

The first step to solve any problem is recognition. While most implementation problems are relatively straight-forward, some may be a bit fuzzy in its complexity. For instance, it is difficult to estimate the time complexity for solving this Sumdoku problem with brute force search and pruning.

Unfortunately, there is not really a good way to determine whether problems like this are solvable with brute force, so it often becomes a contest strategy to decide whether or not to attempt it. Nevertheless, given that US contests have a tendency to feature these kinds of problems and one can be implemented in under 30 minutes, it often is reasonable to attempt a brute force on these problems.

3.2 Encoding & Bitmask

Imagine you have an implementation task that deals with poker hands. While it is tempting to implement a struct for each card that consists of a suit and a rank, you will also have to implement initializer, comparer, etc. which greatly increases code length. Instead, we can encode every card to an int:

```
card_int = (rank << 2) + suit;
```

3.3 Separate Data from Code

Following the same example of poker hands, when we are trying to convert a card from a string (e.g. CK, DJ, H9, etc.) to the encoding, it makes sense to treat these characters as data rather than code logic, i.e.

```
const char suits[5] = "CDHS";
const char ranks[14] = "23456789TJQKA";

int str_to_int(const std::string &str) {
    return (std::find(ranks, ranks + 14, str[1]) - ranks << 2) +
           (std::find(suits, suits + 5, str[0]) - suits);
}

std::string int_to_str(int enc) {
    return std::string(1, suits[enc & 3]) + std::string(1, ranks[enc >> 2]);
}
```

3.4 General Game Theory

> **Sample Problem:** Endgame
> **Link:** `https://vjudge.net/problem/Baekjoon-19350`

A common topic in implementation tasks is to figure out how to win in various board games. At the first glance, these problems seem like trivial searching problems. However, often the cases are complicated by the fact that there is a way for players to achieve a loop: constantly shuffling their pieces back and forth, etc, so that the game never ends. In these scenarios, a little knowledge on game theory can help a lot.

Let us consider the board and the current player $S = (B, p)$ as a state, and define the state to be *winning* if the current player p has a way to win. Similarly let us define S to be *losing* if p loses no matter what he moves. Let us add a directed edge (S_1, S_2) if there is a way for p to move in S_1 that ends the state in S_2 (obviously S_1 and S_2 have different current players). Let us also define a predetermined set of final winning states W and final losing states L (such as checkmating or being checkmated). Then the following three rules hold:

1. A state S is winning iff $S \in W$ or S has an edge that leads to a losing state.
2. A state S is losing iff $S \in L$ or all edges in S leads to winning states.
3. Otherwise, S is neutral (neither winning nor losing). In this scenario, neither player can win with optimal play.

The first two rules are pretty straightforward, since winning means the ability to get to a state where the opponent is losing, and losing means the inability to get out of a state where the opponent is winning. The third rule is a little fuzzy, but it suffice to notice that S only leads to neutral and (possibly) winning states for the opponent. Since there is no incentive to let the opponent win, p will go to some neutral state S', and his opponent cannot improve the scenario either, since S' also only leads to neutral and (possibly) winning states for p.

With these rules it is possible to design some iterative algorithm that searches the whole space to determine winning and losing states. We will start with some predetermined W and L. Observe that if some state S is losing, all states that leads to S immediately become winning (due to rule 1). And if all states that S leads to (i.e. (S, S')) have been determined and S is still not winning, then S must be losing, since all these states are winning (otherwise S would be winning according to the previous statement). Therefore, we can keep track of for every state S, how many states that it leads to are still undetermined, and update S once one of them become losing or the value becomes zero.

A pseudo-code for the algorithm can be find below:

```
determined_state = {W, L};
queue = {W, L};

while (stack is not empty) {
```

```
    q = top of queue;
    pop q out of queue;
    for (all states s that leads to q) {
        if (s is not in determined_state) {
            if (q is winning) {
                --possible leads of s;
                if (possible leads of s == 0) {
                    s is losing (rule 2);
                    add s to determined_state;
                    add s to stack;
                }
            } else {
                s is winning (rule 1);
                add s to determined_state;
                add s to stack;
            }
        }
    }
}
```

3.5 Careful & Clever Discussion

> **Sample Problem:** Alice and Bomb
> **Link:** `https://vjudge.net/problem/Baekjoon-13801`

This problem represents one of the harder geometry problems you will face in this course. Many aspects of it are informative for other problems, so let us try to break it down.

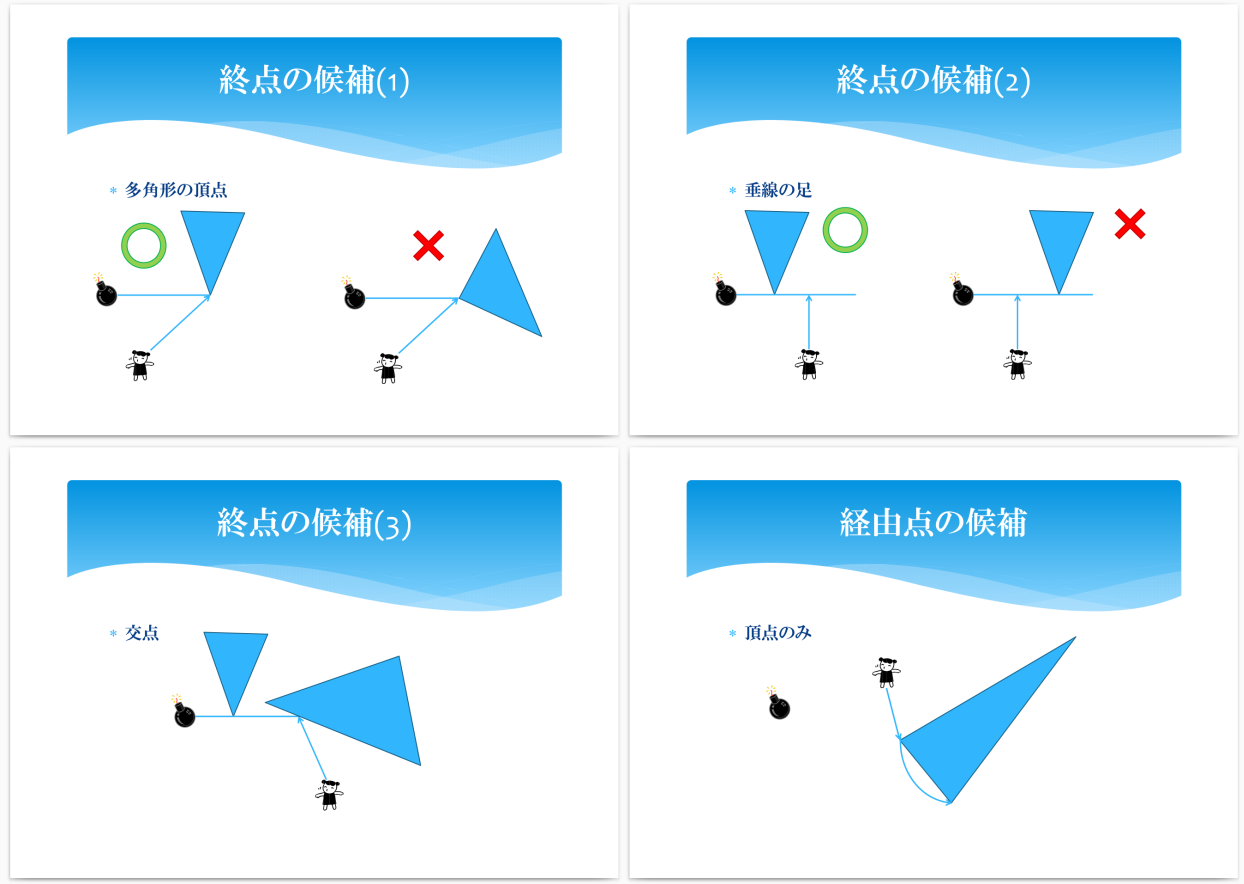
First, intuitively Alice must go to some point that is outside the blast range with the shortest path. Given that the plane consists of polygon barricades, it seems obvious that for the shortest path, Alice will walk in straight segments visiting some vertices of polygons and finally reach her destination.

Next let us consider what kind of destinations there are. Obviously vertices of polygons are possible destinations. It may takes a while to come up with but with sufficient time we should also notice that projection of Alice's positions onto blast lines and the intersection of blast lines and polygon edges are also potential destinations (see the figure below for a detailed description).

All that remains for us is to expand our ideas into a detailed algorithm:

1. Find all possible destinations.
   (a) Alice's current position (i.e. she does not move at all).
   (b) Find all vertices of polygons (trivial, size 500).
   (c) Find all projections of Alice's position onto blast lines (basic geometry, size 500).
   (d) Find all intersections of blast lines and edges (basic geometry, size 500×500)
2. Determine if these destinations are feasible. It is sufficient to check:
   (a) if the segment from the bomb to the destination intersects with something at a point other than the destination (basic geometry).
   (b) if, when standing at the bomb and looking at the destination, there are two segments intersecting at the destination and they are both on the left (right) side. This helps us filter out the polygon vertex case (basic geometry, det product should be sufficient).

   The overall complexity should be 500×500 points and 500 checks for each point.
3. Determine if it is possible to go from vertex u to vertex v in a straight line. This can be done by checking if there is an obstacle edge in the way. (basic geometry, $500 \times 500 \times 500$ complexity)
4. Run a Dijkstra to find the shortest path from start to some feasible destination.

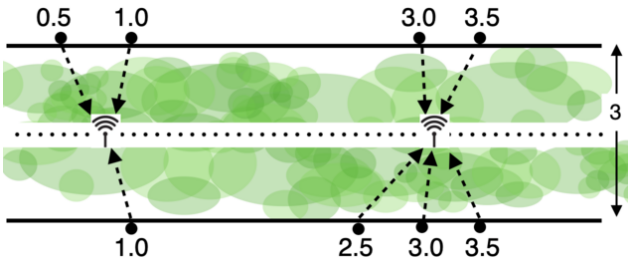


If you follow through the algorithm, you will see that not one single step in this algorithm is hard to come up with, except that we need to come up with three scenarios of possible destinations. Indeed, in these kinds of problems, the main difficulty is to discuss every situation thoroughly.

Note: If you need a refresher on basic geometry, a good start is to take some template (mine is here: `https://github.com/zhtluo/LMR`) and run through this problem list to understand how to use it: `https://judge.u-aizu.ac.jp/onlinejudge/finder.jsp?course=CGL`

B Road Band

All the residents of the rural community of Axes Point live on one of two parallel streets separated by a band of green park land. Recently, the local board of supervisors received a grant to (finally) bring wireless service to the town. The grant provides enough money for them to install k access points, and the supervisors have decided to place them in a straight line on County Road "B," which lies in the wooded band midway between the two residential streets. They want to place them in a way that minimizes the distance between users and their nearest access point. Specifically, they want to minimize the sum of the squares of the distances of each user from their nearest access point. For instance, Figure 1 shows two streets with eight customers and their locations along the streets (this is the first sample input). The streets are 3 units apart, and two access points have been placed at points midway between the two streets so that the sum of the eight squared distances is minimized.



Given the locations of all customers along each of the two streets, the distance between the streets, and the number of access points, help the local government determine the minimum sum of squared distances that can be achieved.

Input

There are three lines of input. The first line contains four integers m, n, k, s, where m and n ($1 \le m, n \le 1\,000$) are the number of customers along each of the two roads, k ($1 \le k \le \min(\max(m, n), 100)$) is the number of access points to be placed, and s ($1 \le s \le 50$) is the distance separating the two roads. The second line contains m floating-point values $x_1, x_2, \ldots, x_m$ ($0 \le x_i \le 1\,000$) giving the locations of the m customers along the first road. The third line is similar, containing n floating-point locations of customers along the second road. All values on each of the second and third lines will be distinct (but some values may appear in both lines). Customer locations will have no more than four decimal places.

Output

Output a single floating-point value equal to the minimum sum of squared distances for each customer from the closest of the k access points. Answers should be correct to within an absolute or relative error of 10^{-5}.

Example

Input

```
4 4 2 3
0.5 1.0 3.0 3.5
1.0 2.5 3.0 3.5
```

Output

```
18.86666667
```

Source

2023-2024 ICPC East North America Regional Contest (ECNA 2023)

Cyclic Substrings

Mr. Ham is interested in strings, especially palindromic strings. Today, he finds a string s of length n.

For the string s of length n, he defines its *cyclic substring* from the i-th character to the j-th character ($1 \le i, j \le n$) as follows:

- If $i \le j$, the cyclic substring is the substring of s from the i-th character to the j-th character. He denotes it as $s[i..j]$.
- If $i > j$, the cyclic substring is $s[i..n] + s[1..j]$, where + denotes the concatenation of two strings. He also denotes it as $s[i..j]$.

For example, if $s =$ `12345`, then $s[2..4] =$ `234`, $s[4..2] =$ `4512`, and $s[3..3] =$ `3`.

A string t is *palindromic* if $t[i] = t[n - i + 1]$ for all i from 1 to n. For example, `1221` is palindromic, while `123` is not.

Given the string s, there will be many cyclic substrings of s which are palindromic. Denote P as the set of all **distinct** cyclic substrings of s which are palindromic, $f(t)(t \in P)$ as the number of times t appears in s as a cyclic substring, and $g(t)(t \in P)$ as the length of t. Mr. Ham wants you to compute

$$\sum_{t \in P} f(t)^2 \times g(t)$$

The answer may be very large, so you only need to output the answer modulo 998 244 353.

Input

The first line contains a number n ($1 \le n \le 3 \times 10^6$), the length of the string s.

The second line contains a string s of length n. Each character of s is a digit.

Output

Output a single integer, denoting the sum modulo 998 244 353.

Examples

Input

```
5
01010
```

Output

```
39
```

Input

```
8
66776677
```

Output

```
192
```

Note

In the sample, the palindromic cyclic substrings of s are:

- $s[1..1] = s[3..3] = s[5..5] =$ `0`.
- $s[2..2] = s[4..4] =$ `1`.
- $s[5..1] =$ `00`.
- $s[1..3] = s[3..5] =$ `010`.
- $s[2..4] =$ `101`.
- $s[4..2] =$ `1001`.
- $s[1..5] =$ `01010`.

The answer is $3^2 \times 1 + 2^2 \times 1 + 1^2 \times 2 + 2^2 \times 3 + 1^2 \times 3 + 1^2 \times 4 + 1^2 \times 5 = 39$.

Source

The 2023 ICPC Asia Hefei Regional Contest (The 2nd Universal Cup. Stage 12: Hefei)

Mathematical Problem

The mathematicians of the 31st lyceum were given the following task:

You are given an **odd** number n, and you need to find n different numbers that are squares of integers. But it's not that simple. Each number should have a length of n (and should not have leading zeros), and the multiset of digits of all the numbers should be the same. For example, for `234` and `432`, and `11223` and `32211`, the multisets of digits are the same, but for `123` and `112233`, they are not.

The mathematicians couldn't solve this problem. Can you?

Input

The first line contains an integer t ($1 \le t \le 100$) — the number of test cases.

The following t lines contain one **odd** integer n ($1 \le n \le 99$) — the number of numbers to be found and their length.

It is guaranteed that the solution exists within the given constraints.

It is guaranteed that the sum of n^2 does not exceed 10^5.

The numbers can be output in any order.

Output

For each test case, you need to output n numbers of length n — the answer to the problem.

If there are several answers, print any of them.

Example

Input

```
3
1
3
5
```

Output

```
1
169
196
961
16384
31684
36481
38416
43681
```

Note

Below are the squares of the numbers that are the answers for the second test case:

$169 = 13^2$

$196 = 14^2$

$961 = 31^2$

Below are the squares of the numbers that are the answers for the third test case:

$16384 = 128^2$

$31684 = 178^2$
$36481 = 191^2$
$38416 = 196^2$
$43681 = 209^2$

Source

Codeforces Good Bye 2023

Forest for the Trees

You have sent a robot out into the forest, and it has gotten lost. It has a sensor that will detect all the trees around itself regardless of any occlusions, but unfortunately in this forest, all trees look alike. You do have a map of all trees in the forest, represented as (x, y) points. Conveniently, since this used to be a tree farm, all trees are at integer coordinates, though not all coordinates are occupied. The robot's sensor tells you the x and y distance to each tree within range, relative to the front of the robot. However, the robot is heading in an unknown direction relative to the map, so each sensor reading is given as a tuple of (distance to the right of the robot, distance forward of the robot) and either value can be negative since the robot can sense in all directions. Helpfully, the robot will always place itself at integer coordinates and aligned to the positive or negative x or y axis, and will never be at the same location as a tree. Can you find out where the robot is?

Input

The first line of input contains three integers: n_t, the number of trees in the forest, n_s, the number of trees sensed by the robot, and r_{max}, the maximum Manhattan distance (sum of x and y distances) of any sensor reading. The next n_t lines each contain two integers representing the (x, y) locations of all the trees relative to a global coordinate system. The final n_s lines each contain two integers. The first integer in the i^{th} sensor reading, $s_{i,x}$, represents the distance to the tree along the axis perpendicular to the robot's heading and the second integer $s_{i,y}$ represents the distance along the axis parallel to the robot's heading. You can assume that $|s_{i,x}| + |s_{i,y}| \leq r_{max}$ for all i. You may also assume $0 < n_t \leq 5000$, $0 < n_s \leq 1000$, $0 < r_{max} \leq 1000$, and all tree locations have x and y coordinates $-100,000 \leq x, y \leq 100,000$.

Output

Print one of the following: the x, y location of the robot, printed as two integers separated by a space; "Impossible" if there is no location in the map that could produce the given sensor values, or "Ambiguous" if two or more distinct locations and/or orientations could produce the given sensor values.

Example

Input

```
4 4 100
1 1
2 2
2 1
3 3
0 1
0 2
-1 2
-2 3
```

Output

```
0 1
```

Source

2023-2024 ICPC East North America Regional Contest (ECNA 2023)

ICPC Ranking

There are so many submissions during this contest. Coach Pang can not determine which team is the winner. Could you help him to print the score board?

As we all know, the contest executes by the teams submit codes for some problems. Simply, we assume there are three kinds of results for every submission.

ERROR There was something wrong. The team didn't solve the problem but won't get any penalty.

NO Sorry, the code was not right. The team didn't solve the problem.

YES Yeah, AC. The team solved the problem.

To make the contest more exciting, we set a time called frozen time. If one team doesn't solve one of the problems before the frozen time and submits at least one submission on this problem after or exactly at the frozen time, this problem of this team is called frozen. For different team, the frozen problems will be different. For the frozen problems, the score board will only show how many submissions the team has been submitted but won't show the result of the submissions.

The rank is determined by the following factors. Remember, we only consider the unfrozen problems. The frozen problems will be ignored. The following factors are ordered with the priority from high to low.

Solved the team who solves more problems will place higher.

Penalty the team who gets less penalty time will place higher. Only solved problems will give penalty time. Every solved problem will give $T + 20X$ penalty time. T is the time of the first YES, X is the number of NOs before the first YES.

Last Solved the team who solved their last problem earlier will place higher. If there is a tie, we compare their second last problems, then their third last problems, etc.

Name The team whose name comes later in lexicographical order will place higher.

At the end of the contest, the score board will be unfrozen. First, choose the team which has frozen problems with the lowest rank. Then choose one frozen problem of this team. If the team has multiple frozen problems, choose their first frozen problem in alphabetic order. Then show the result of the problem, recalculate the rank, change the score board and make the problem unfrozen for this team. Repeat this procedure until no teams have frozen problems. Then we get the final score board.

Please help Coach Pang to print the initial score board, the final score board and the process of the unfreeze procedure.

Input

The first line contains an integer C, which indicates the number of test cases.

For each test case, the first line will have four integers n, m, T and t. n ($1 \le n \le 50000$) is the number of submissions. m ($1 \le m \le 26$) is the number of problems. T ($1 \le T \le 10000$) is the total time of the contest. t ($0 \le t \le T$) is the frozen time.

The following n lines, each line will be in the form "Name Problem Time Result". Name is the team's name which only contains letters and digits with at most 20 characters. Problem is the identifier of the problem which is a capital letter from A to them-th letter. Time is a integer indicates the submission time which greater than or equal to 0 and less than T. Result is one string which equal YES, NO or ERROR.

Every team will have at least one submission. If one team has multiple submissions at the same time, we consider the ERRORs come before NOs and NOs come before YESs.

Output

For each test case, first print "Case #x:", x is the case number start from 1.

Then print the initial score board (before the unfreeze procedure) ordered by the rank. For every team, print "Name Rank Solved Penalty A B C ...". Name is the team's name. Rank is the team's rank. Solved is the number of solved problems. Penalty is the team's penalty time.

A B C ... is the condition of every problem is the format below:

+x The problem is unfrozen and solved. x is the number of NOs before the first YES. If x = 0, print "+" instead of "+0".

-x The problem is unfrozen but not solved. x is the number of NOs. If x = 0, print "." instead of "-0".

-x/y The problem is frozen. x is the number of NOs before the frozen time. y is the number of submissions after or exact at the frozen time. If x = 0, print "0/y" instead of "-0/y".

After the initial score board, print the process of the unfreeze procedure. During the unfreeze procedure, every time one team unfreezes one frozen problem and causes the change of its rank, print "Name1 Name2 Solved Penalty". Suppose this team is team A. Name1 is the name of team A. Name2 is the name of the team which is overtaken by team A with the highest rank. Solved is the new number of the solved problems of team A. Penalty is the new penalty time of team A.

Finally print the final score board (after the unfreeze procedure) as the same format as the initial score board.

See the sample to get more details.

Example

Input

```
20 12 300 240
Epic B 12 YES
Epic A 14 NO
Rivercrab E 25 YES
Two2erII B 100 NO
Epic A 120 YES
Rivercrab I 150 NO
Two2erII C 160 NO
Epic C 180 YES
Two2erII C 180 NO
Rivercrab F 226 YES
Two2erII C 230 YES
Two2erII L 241 YES
Epic F 246 YES
Epic G 260 YES
Rivercrab I 289 YES
Epic D 297 YES
Musou H 299 YES
Musou I 299 YES
Musou J 299 YES
Musou K 299 YES
```

Output

```
Case #1:
Epic 1 3 332 +1 + + 0/1 . 0/1 0/1 . . . . .
Rivercrab 2 2 251 . . . . + + . . -1/1 . . .
Two2erII 3 1 270 . -1 +2 . . . . . . . . 0/1
Musou 4 0 0 . . . . . . . . 0/1 0/1 0/1 0/1 .
Musou Two2erII 2 598
Two2erII Musou 2 511
Musou Rivercrab 3 897
Rivercrab Musou 3 560
Musou Epic 4 1196
Epic Musou 4 629
```

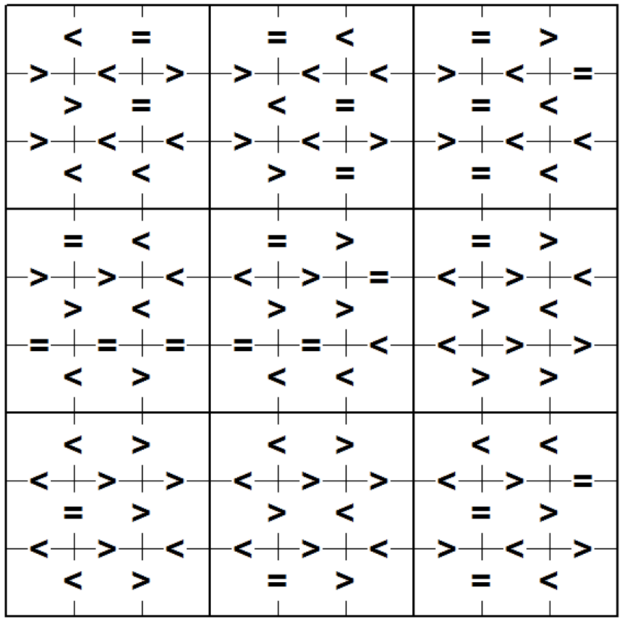
```
Epic 1 6 1135 +1 + + + . + + . . . . .
Musou 2 4 1196 . . . . . . . + + + + .
Rivercrab 3 3 560 . . . . + + . . +1 . . .
Two2erII 4 2 511 . -1 +2 . . . . . . . . +
```

Source

2013 Asia Chengdu Regional Contest

Sumdoku

Sumdoku is a variant of the game Sudoku. As in Sudoku, the aim is to fill in a 9-by-9 grid with the digits 1 through 9 so that each digit 1 through 9 occurs exactly once in each row, exactly once in each column, and exactly once in each of the 9 3-by-3 sub-squares subject to constraints on the choices. In Sudoku, the constraints are that certain squares must contain fixed values. In Sumdoku, the constraints are on the sum of adjacent squares within each 3-by-3 sub-square. In the illustration below, the symbols <, =, and > indicate that the sum of the values on either side (or above and below) the symbol must have a sum less than 10, equal to 10, or greater than 10, respectively.



Write a program to solve Sumdoku problems.

Input

The first line of input contains a single decimal integer P, ($1 \leq P \leq 10000$), which is the number of data sets that follow. Each data set should be processed identically and independently.

Each data set consists of 16 lines of input. The first line contains the data set number, K. The following 15 lines consist of the characters <, =, or >. Rows 1, 3, 5, 6, 8, 10, 11, 13, and 15 contain 6 characters corresponding to constraints on the sum of values to the left and right of the symbol. Rows 2, 4, 7, 9, 12, and 14 contain 9 characters corresponding to constraints on the sum of values above and below the symbol. Note: Solutions to some problems may not be unique. The judging program will just check whether your solution satisfies the constraints of the problem (row, column, 3-by-3 box, and inequality constraints).

Output

For each data set, there are 10 lines of output. The first output line consists of the data set number, K. The following 9 lines of output consist of 9 decimal digits separated by a single space. The value in the jth position in the ith line of the 9 output lines is the solution value in column j of row i.

If there are multiple solutions, print the lexicographically smallest one if the answer is read by row-major order.

Example

Input

```
1
1
<==<=>
><>><<><=
>=<==<
><<>><>><<
<<>==<
=<=>>
>><<>><><
><>>>><
=====<><>
><<<>>
><>><<
><><><><>=
=>><=>
><<<><><><
><>=>=<
```

Output

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

Source

2017 Greater New York Programming Contest

Endgame

The game of chess is almost finished. On the chessboard, apart from White and Black kings, there is only a White rook.

You are playing White, and it is your move. Determine the minimal number of moves you need to give a checkmate, provided that your opponent plays optimally and delays his inevitable defeat for as long as possible.

There is a compilation of chess rules at the end of this statement. If you already know them, rest assured: your puny chess skills will not help you solve this problem.

Input

The first line of input contains the number of test cases z ($1 \leq z \leq 10$). The descriptions of the test cases follow.

Each test case is given on eight lines describing a chessboard. Each of these lines describes a single row and contains exactly eight characters: '.' denotes an empty field, 'W' is the White king, 'B' is the Black king, and 'R' is the White rook. There is exactly one piece of each kind. The starting position is guaranteed to be valid: in particular, kings are not adjacent to each other, and the Black king is not under attack.

There is an empty line after each test case.

Output

For each test case, output a line containing a single integer: the maximal possible number of moves White needs to give a checkmate (per common tradition, count only your moves, not Black's).

Example

Input

```
2
........
........
........
........
........
.......W
R.......
.......B

....B...
........
..W.....
........
.....R..
........
........
........
```

Output

```
1
2
```

Hints

Chess rules:

1. The players alternately move one piece per turn.
2. A player cannot “pass”; on each turn, they have to make a legal move.
3. The king moves one square in any direction (horizontally, vertically, or diagonally).
4. The rook can move any number of squares along any row or column, but may not leap over other pieces.
5. A king is under attack if it is within move range of an opposing piece.
6. A player may not make any move that would put or leave his or her king under attack (in particular, the king cannot be moved to a square adjacent to other king).
7. A Black king can, however, move to a square occupied by the White rook, if the White king is not adjacent to the rook. The rook is then captured and the game ends in a draw.
8. If Black player has no legal move, the game is over; it is either a checkmate (White wins) if the Black king is under attack, or a stalemate (a draw) if it is not.
9. It is known that, in the situation described above (king and rook vs. king), a checkmate is always possible in less than 50 moves.

Source

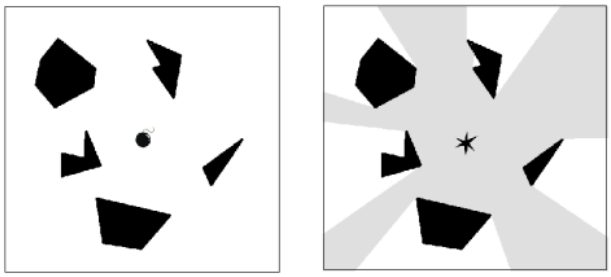
Petrozavodsk Programming Camp, Winter 2017, Day 1: Jagiellonian U Contest

Alice and Bomb

Alice and Bob were in love with each other, but they hate each other now. One day, Alice found a bag. It looks like a bag that Bob had used when they go on a date. Suddenly Alice heard tick-tack sound. Alice intuitively thought that Bob is to kill her with a bomb. Fortunately, the bomb has not exploded yet, and there may be a little time remained to hide behind the buildings.

The appearance of the ACM city can be viewed as an infinite plane and each building forms a polygon. Alice is considered as a point and the bomb blast may reach Alice if the line segment which connects Alice and the bomb does not intersect an interior position of any building. Assume that the speed of bomb blast is infinite; when the bomb explodes, its blast wave will reach anywhere immediately, unless the bomb blast is interrupted by some buildings.

The figure below shows the example of the bomb explosion. Left figure shows the bomb and the buildings (the polygons filled with black). Right figure shows the area (colored with gray) which is under the effect of blast wave after the bomb explosion.



Your task is to write a program which reads the positions of Alice, the bomb, and the buildings and calculate the minimum distance required for Alice to run and hide behind the building.

Input

The input contains multiple test cases. Each test case has the following format:

Input

The input contains multiple test cases. Each test case has the following format:

$$
\begin{aligned}
&N \\
&b_x\ b_y \\
&m_1\ x_{1,1}\ y_{1,1}\ \cdots\ x_{1,m_1}\ y_{1,m_1} \\
&\vdots \\
&m_N\ x_{N,1}\ y_{N,1}\ \cdots\ x_{N,m_N}\ y_{N,m_N}
\end{aligned}
$$

The first line of each test case contains an integer N ($1 \leq N \leq 100$), which denotes the number of buildings. In the second line, there are two integers b_x and b_y ($-10000 \leq b_x, b_y \leq 10000$), which means the location of the bomb. Then, N lines follows, indicating the information of the buildings.

The information of building is given as a polygon which consists of points. For each line, it has an integer m_i ($3 \leq m_i \leq 100$, $\sum_{i=1}^{N} m_i \leq 500$) meaning the number of points the polygon has, and then m_i pairs of integers $x_{i,j}$, $y_{i,j}$ follow providing the x and y coordinate of the point.

- Alice is initially located at $(0, 0)$.
- $N = 0$ denotes the end of the input.

Output

For each test case, output one line which consists of the minimum distance required for Alice to run and hide behind the building. An absolute error or relative error in your answer must be less than 10^{-6}.

Example

Input

```
1
1 1
4 -1 0 -2 -1 -1 -2 0 -1
1
0 3
4 1 1 1 2 -1 2 -1 1
1
-6 -6
6 1 -2 2 -2 2 3 -2 3 -2 1 1 1
1
-10 0
4 0 -5 1 -5 1 5 0 5
1
10 1
4 5 1 6 2 5 3 4 2
2
-47 -37
4 14 3 20 13 9 12 15 9
4 -38 -3 -34 -19 -34 -14 -24 -10
0
```

Output

```
1.00000000
0.00000000
3.23606798
5.00000000
1.00000000
11.78517297
```

Source

JAG Practice Contest 2010