VCU Computer Science
2024 High School Programming Contest

There are ten problems (A-J) to solve.
The problems are not sorted by difficulty.

Problem	Problem Name	Problem Color
A	Carrots	Black
B	Robot Spread	Green
C	Induction Heads	Grey
D	Quantum Swaps	Orange
E	Shy Spies	Pink
F	Compute Rush	Purple
G	Alchemy	Red
H	Maize	Royal Blue
I	Ants	Teal
J	Ingredients	Yellow

Problem A
Carrots

Jasper Rabbit loves his carrot patch. It has been in his family for generations. In each year, the patch of carrots expands a bit; specifically, its area grows by 0.1%, that is, it is 1.001 times larger in terms of area than the year before. Originally, Jasper's grand-grand-...-grand-grandpa started the patch with 1 square foot in area. After 1 year, it spread to be 1.001 sq ft. Currently, after 700 years, the area is about 2.01 square feet. Jasper, being philosophically inclined, often wonders how large the patch will be by the time of his grand-grand-...-grand-grandchildren. Can you help him figure that out?

Input

First line will contain a single integer number n, ($1 <= n <= 10000$) the number of years the patch will grow, at a rate of 1.001 per year, from the original area of 1 sq ft.

Output

Print one line containing a single number, the total area of the patch after n years, rounded down to the nearest integer.

Sample Input

700

Sample Output

2

Problem B
Robot Spread

The Robot Spread board game is taking Byteland by storm. It is a unique game where you have a board in a form of a grid, R by C cells, and every cell has its unique weight in the game. A player at the start of the game has one robot located at their chosen starting position, at cell (r[i],c[j]). In one unit of time a robot can build other robots on all empty cells within a distance less than or equal to K steps from its current cell; in one step a robot can only visit its 8 neighboring cells of its current cell. For example, if we have a robot in cell (3,3) on a 5-by-5 board, and K = 1, then, this robot in only one unit of time will construct robots in cells (2,2) , (2,3) , (2,4) , (3,2), (3,3) , (3,4) , (4,2) , (4,3) and (4,4), assuming these cells are empty. After that one unit of time we will have 9 robots: the original one at (3,3), and eight new ones at the neighboring cells.

At the end of the game, each robot will have a score based on the weight of its current cell. The player's final score is determined as the sum of all his robot's scores. The trick of this game is to find the ultimate score that you will be able to collect after T units of time based on the location of your first robot.

Given the size of the grid, the maximum distance K and the row and column of the first robot's cell, and T units of time, you need to find the maximum score that you will get after these T units of time. For example, for a 3-by-3 board with weights as below:

1	2	1
4	3	3
4	***2***	1

if the staring point is row 3 and column 2 (in bold italics above), and K=1, T=1, in the end there will be robots in six cells (the shaded area), and the maximum score will be 4+3+3+4+2+1 = 17.

Input

The first line contains the number of test cases, Z (1<=Z<=100). Individual test cases follow.

For each test case, the first line contains four integers: R the number of rows, C the number of columns, K the maximum distance and T total available units of time. (1 =< R,C,K,T <= 100)

The next line contains two integers *r* (1<= *r*<= *R*) and c (1<=*c*<=*C*) the position of the first robot.

Each of the next *R* lines contain *C* integers, S[i,j] representing the weights of each cell (row i, column j) in the grid, where 1<= S[i,j] <= 10^9

Output

For each test, print the answer modulo 10^9 +7

Sample Input

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

Sample Output

```
17
21
```

Problem C
Induction Heads

Computer scientists in Byteland recently trained a new AI model: Gigantic Prefix-based Text-mutator (or, GPT, for short). The model takes text with n words on input, and produces new text with n words on output. It operates iteratively, in stages called *layers*. The first layer reads the input text (one word at each position, that is, an array *input*[1,...,n] of words), and produces a new text (n words, array *output*[1,...n]). This new text is then read by the next layer of the model, which produces another new text. The process continues until last, k-th layer – the *output*[1,...,n] array of words that layer produces is the output text of the whole GPT model.

Each layer produces its output text by taking words from the text the layer received on input. But how exactly the words are selected remained a mystery to the scientists of Byteland. Only recently, they observed that what each layer has learned to do is actually very simple. First, a layer just copies input to the output, word by word. Then, it sometimes modifies the output based on layer's input. It treats input words as pairs, at odd and the subsequent even positions (e.g. *input*[3] and *input*[4]); if last word of input is at odd position, it is not part of any pair. A layer goes through pairs from left to right (input indices (1,2), (3,4), ...), if it sees a copy of the odd input word from a pair somewhere to the right of it in the input, it will replace it with the even input word from that pair – but it will replace it the output, without affecting the input.

For example, if input to a layer is five-word text "*sweet home home sweet home*", the layer will first make the output to be "*sweet home home sweet home*", then it will change "*sweet*" at position 4 in the output to "*home*", using odd-even pair ("*sweet home*") from input positions (1, 2), and then it will replace "*home*" at position 5 in the output to "*sweet*", using odd-even pair ("*home sweet*") from input positions (3, 4). The final output of the layer will be "*sweet home home home sweet*".

The scientists named this behavior "induction head", because it allows GPT to perform a form of simple reasoning and provide some useful answer. For example, if you want to know what "llama" is, and have a collection of facts, some relevant some not, like: *llama is a camelid*, *camelid is a mammal*, *mammal is an animal*, *grass is a plant*, etc., you can form the input to GPT by first listing the facts as odd-even pairs and then, at the end, adding your query (i.e., "*llama*"), like this:

llama camelid camelid mammal mammal animal grass plant llama

The first layer will then produce a new text on output:

llama camelid camelid mammal mammal animal grass plant camelid

Then, second layer will take that as input, and produce

llama camelid camelid mammal mammal animal grass plant mammal

The third layer will take that as input, and produce

llama camelid camelid mammal mammal animal grass plant animal

Now we can read the answer from the query position: *llama* is an *animal*.

Scientists discovered that the order in which the facts are listed in the input matters, e.g. for

grass plant mammal animal camelid mammal llama camelid llama

the first layer will produce:

grass plant mammal animal camelid animal llama mammal camelid

then, second layer will take that as input, and produce

grass plant mammal animal camelid animal llama animal animal

and there is no need for a third layer! The answer that *llama* is an *animal* is already there!

Your task is to help the scientists study this phenomenon further. Given an input text with a query at the last, odd position, and the known true answer (*animal* in the text above), calculate what is the smallest number of GPT layers needed to have that answer produced at the last position of the text. For simplicity, instead of words, you will just use numbers (e.g. *animal*->1, *camelid*->2, *grass*->3, *llama*->4, *mammal*->5, *plant*->6), so, for example, input

llama camelid camelid mammal mammal animal grass plant llama

will be just

4 2 2 5 5 1 3 6 4

and the desired answer will be 1 instead of *animal*.

Input

First line will contain a single integer number *T (1<=T<=100)*, the number of test cases that follow. Each test case will consist of three lines. First line will consist of a single integer *n* (3 <= *n* <= 99, *n* odd), the length of the input text for GPT. The second line will consist of a single integer *k* (1 <= *k* <= 1000), the word number of the correct answer. The third line will consist of *n* integers (each from the range [1,...,1000]) separated by spaces – this is the input sequence for the GPT encoded as numbers instead of words.

Output

Print *T* lines, one per each test case. In each line, print the smallest number of GPT layers needed for the correct answer to be produced by GPT for the input for that test case. Print 0 if the true answer cannot be produced by GPT no matter how many layers are used.

Sample Input

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

Sample Output

```
3
2
```

Problem D
Quantum Swaps

In quantum computing, instead of bits, we have quantum bits, known as qubits. Qubits are physical devices, and are often arranged in a quantum computer according to a qubit connectivity graph, where qubits are vertices, and edges represent which qubits are neighbors. Quantum computers perform operations on the information stored in sets of qubits. Currently, quantum computers can only perform an operation `op(qa,qb,qc,...)` on a set `(qa,qb,qc,...)` of qubits if all the qubits involved in the operation are each other's neighbors. For example, if four qubits are arranged in a linear graph `q1-q2-q3-q4`, only some two-qubit operations are possible to execute: `op(q1,q2)`, `op(q2,q3)`, etc., But `op(q1, q4)` is not possible since `q1` and `q4` are not neighbors. Also, `op(q1,q2,q3)` is not possible, since `q1` and `q3` are not neighbors, even though they are both neighbors of `q2`.

One way of dealing with the limitation caused by the neighborhood structure of the qubit connectivity graph is to use two-qubit SWAP quantum operation, which swaps the values that two qubits store. For example, if five qubits `q1-q2-q3-q4` together store a binary string `0101` (`q1` has value 0, `q2` has value 1, `q3` has value 0, etc.), after `SWAP(q1,q2)` operation, `q1` will have value 1 while `q2` will value 0, and the qubits together will be storing binary string `1001`. Of course, like any quantum operation, `SWAP` can only work on two qubits that are neighbors; for example, `SWAP(q1,q3)` is not possible for qubits arranged as `q1-q2-q3-q4`.

With the help of `SWAP`, executing `op(q1,q3)` becomes possible in a linear qubit connectivity graph, as a series of two two-qubit operations involving neighbors:

```
SWAP(q1,q2) # qubits q1 and q2 swap content
op(q2,q3) # after swap, this is now equivalent to op(q1,q3)
```

We just need to remember that the outcome of the op will be in qubits `q2` and `q3`, not in `q1` and `q3`.

Similarly, a two-op program:

```
opA(q2,q3)
opB(q1,q4)
```

can become possible to execute on a linear-topology quantum computer if we place two SWAP operations between the two ops:

```
opA(q2,q3)
SWAP(q1,q2)
SWAP(q4,q3)
opB(q2,q3)
```

We just need to remember that when we read the results after the computations, to get the resulting bitstring we should read bits from qubits in order `q2,q1,q4,q3`, instead of the original order `q1,q2,q3,q4`.

One of the existing problems in quantum computing is: given a qubit connectivity graph and a series of ops (each operating on some qubits), design a series of `SWAP` operations that can be placed in between these ops, to make the ops possible to execute on a given qubit connectivity graph.

Your task is to help solve this problem for a simple qubit connectivity graph: a linear graph with up to 100 qubits, and for a simple type of programs, those where each op involves two different qubits (that is, no qubit is involved in more than one op in the original program). Specifically, given a sequence of *m* operations (*m*<=100), that is, a sequence of *m* non-intersecting pairs of qubits, calculate the smallest number of `SWAP` operations that need to be inserted in order to make the quantum program possible to execute. It is ok if the results are in different qubits than the original order (as in examples above).

Input

First line will contain a single integer number *T (1<=T<=100)*, the number of test cases. Each test case, on the first line, will contain one integer *m* (1<=*m*<=100), the number of operations. Then, *m* lines will follow, each containing one operation, specified as two numbers (each between 1 and 100) indicating the position (in the linear graph of qubits) of qubits involved in the operation.

Output

Print *T* lines, one per test case. For each test case, print one integer: the smallest number of `SWAP` operations need to make the program executable.

Sample Input

```
2
3
6 5
4 3
2 1
2
1 3
2 4
```

Sample Output

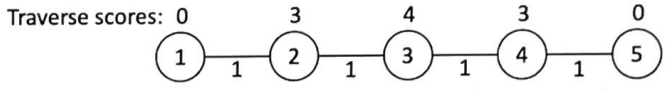
```
0
1
```

Problem E
Shy Spies

Byteland is sending a spy to Bitworld: it wants to keep tabs on what happens in there, while remining undetected. The Master of Secrets of Byteland formed a plan on how to achieve that goal: given the map of major cities of Bitword, with distances of the major roads connecting them, the spy should keep tabs on a city that is most traversed. But instead of establishing a hideout there, the spy should establish their base in a city that is least traversed among the cities that are direct neighbors of the most traversed city. It may turn out that there are several equally most-traversed cities – then the idea is to spy on at least one of them, by looking at neighbors of all of them and finding least traversed city among them. It may also turn out that among the neighbors, there are several equally least-traversed cities; if so, the Master of Secrets want to know all of these least traversed cities before making the decision on the location.

How much a given city X is traversed is assessed in the following way. For each pair of cities not including X, the number of shortest paths between these pair of cities that pass through X is calculated, and divided by the total number of shortest paths between the pair (that is, including shortest paths that do not pass through X). This number is summed up over all pairs of cities not including X, and the result of the summation becomes the *traverse score* for city X. Most traversed city/cities from a set of cities are those with *traverse score* equal to the maximum score in the set. Similarly, given a set of cities, least traversed city/cities are those with *traverse score* equal to the minimum score in the set.

For example, in a simple, linear graph with five vertices, 1,...,5, connected using undirected edges with weight 1, vertex 3 has the highest traverse score, 4 (it is on all shortest path between 0 and 4, all shortest paths between 0 and 5, and the same for 1 and 4, and 1 and 5).



In this graph, vertices 2 and 4 (both with traverse scores of 3) are the solution.

Input

First line will contain a single integer number *T*, the number of test cases (1<=*T*<=100). Each test case starts with a line containing two integers, *n* and *m*, the number of cities and the number of direct roads, respectively (3<=*n*<=100, 2<=*m*<=1000). Then, *m* lines follow, each consisting of three integers: the ID's of two cities connected by a bi-directional road (each number in range 1,...,*n*), and the distance (an integer in range 1,...,1000).

Output

Print *T* lines, one per test case. On each line, print ID's of all cities in which the spy should establish the base, for that test case.

Sample Input

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

Sample Output

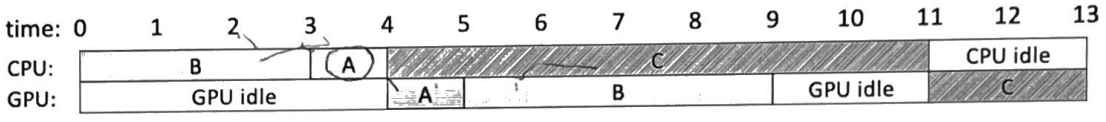
```
2 4
```

Problem F
Compute Rush

Modern large AI models run on GPU, but often require some substantial processing of input data on CPU before computing can be sent to GPU. You have a single machine with one CPU and one GPU, and you want to run a series of CPU-GPU AI jobs. For each job, you know how long the CPU processing will take, and how long the GPU processing will take. Can you find the shortest time it would take to process all the AI jobs?

A CPU part of a job utilizes the CPU fully, and cannot be interrupted: once it is started, it needs to finish. Similarly, a GPU part of a job utilizes the GPU fully, and cannot be interrupted. But the GPU parts of AI jobs do not have to be processed on the GPU in the same order as the CPU parts were processed on the CPU, the only constraint is that a GPU part can only start once the CPU part ends; it can start immediately, or any time later.

For example, given 3 AI jobs, with processing times:
Job A: CPU, 1 second; GPU, 1 second
Job B: CPU, 3 seconds; GPU, 4 seconds
Job C: CPU, 7 seconds; GPU, 2 second
The shortest time they can all be processed is 13 seconds, it can be achieved for example by the following scheduling of CPU and GPU parts of the jobs:



Input
First line will contain a single integer number *T*, the number of test cases (1<=*T*<=100). Each test case starts with a line containing one integer, *n* (1<=*n*<=100), the number of jobs. Then, n lines follow, each consisting of two integers: the number of CPU seconds needed, and the number GPU seconds needed to complete the job (both numbers in 1,...,100) range.

Output
Print *T* lines, one per test case. On each line, print one integer, the earliest time (in seconds) since the start of the first job on CPU, in which all jobs are completed.

Sample Input
```
1
3
1 1
3 4
7 2
```

Sample Output
```
13
```

Problem G
Alchemy

Alchemists of Oreland know many ways to transmute (that is, change) metals into other metals. These transformations do not follow conventional physics, and can substantially alter the mass, without producing any devastating side-effects. For example, one alchemist knows how to transmute 100oz of lead into 10oz of gold. Another one knows how to change 100oz of gold into 20oz of lead; he rarely has the opportunity to put that knowledge into practice, though.

Some of the metals are finicky – they can be changed into something else only when in a specific state of matter. For example, lead can be turned into gold, but only if the lead is liquid (the resulting gold will also be liquid); no spells will work on solid lead. That does not seem to be much of a technical challenge – furnaces and ice baths are abundant in Oreland – though it does affect the profits: changing state from liquid to solid, or solid to liquid, involves a slight loss of mass; one gets 99oz of output for 100oz of input, for any metal. For example, starting from 100oz of solid lead, one gets 99oz of liquid lead, then 9.9oz of liquid gold, and only 9.801oz of solid, ready-to-ship, gold.

As you could imagine, the ability to change cheap metals into more expensive ones turned out to be quite profitable for the alchemists of Oreland, and apprentices flocked into the profession. One of them, one day, surprised the guildsmen with a simple yet profound question: do we need to keep paying for, say, iron in order to keep selling gold? Yes, seemed the obvious answer... yet some kept thinking... Their deliberations soon coalesced on a more specific question: can we, via a cycle of transmutations, turn 100oz of gold into, say, 102oz of gold, sell the extra 2oz, and repeat? Or, at least turn 100oz of lead or some other metal into 102oz of that metal, so we don't need to ever buy any ingredients again?

Alas, answering that question was beyond their skills. They turned to you, an algorithmics sage, and promised you great wealth if you can settle the matter once and for all. You asked them for the details of all the transmutations they have the abilities to perform (how much metal X results in how much of metal Y, which metals need to be in liquid form to be transformer, which need to be solid, and for which it doesn't matter), and now you need to figure out the answer: is it possible to find a metal, such that starting with 100oz of it (either liquid, or solid), through a series of transmutations, we can obtain more than 100oz of it, in the same state of matter as that start?

For example, if we only have two metals (1 and 2), and metal 1 needs to be in liquid form to be transmuted, while metal 2 needs to be solid, and 100oz of metal 1 can be transmuted to 105oz of metal 2, while 100oz of metal 2 can be transmuted to 98oz metal 1, the answer is yes: 100oz of metal 1 in liquid state can be transmuted into 105oz of metal 2 in liquid state, which can be turned into solid state but with a 1% loss of mass, resulting in 103.95oz of solid metal 2. Solid metal 2 can be turned into solid metal 1 at 100-to-98 rate, yielding 101.871oz of solid metal 1. Turning it into liquid metal 1, with 1% loss of mass, leads to 100.85229oz of liquid metal 1: that is more than the 100oz at the start of the process, thus, the answer is yes.

Input

First line will contain a single integer number T, the number of test cases ($1<=T<=100$). Each test case starts with a line containing four integers: n ($2<=n<=100$), the number of metals; m ($2<=m<=5000$), L ($0<=L<=n$), the number of metals that can only be transformed into something else when liquid, and R ($0<=R<=n$), the number of metals that can only be transformed into something else when solid. We will have $L+R <= n$. If $L>0$, the next line contains a list of L metals IDs, the metals that need to be in liquid form to be transformed. If $R>0$, the next line contains a list of R metal IDs, the metals that need to be in solid form to be transformed. Finally, the last m lines contain the metal-into-metal transformations, each listed using three integers, u v w, separated by single spaces: the line indicates that metal u can be transformed into metal v, and that for 100oz of metal u, we obtain w oz of metal v ($10<=w<=1000$).

Output

Print T lines, one per test case. On each line, print YES if it is possible to start with 100oz of some metal in a specific state, and after some transformations, end up with more than 100oz of that same metal in that specific state; otherwise print NO.

Sample Input

```
1
2 2 1 1
1
2
1 2 105
2 1 98
```

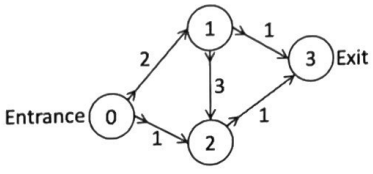
Sample Output

```
YES
```

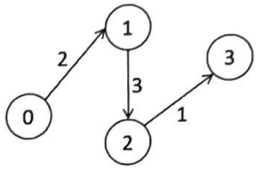
Problem H
Maize

The State Fair has designed a new attraction: a maze. The maze is full of paths cut in a tall cornfield. These paths merge and diverge at junctions; we call each path section between one junction and another a separate corridor. You enter the maze at a single starting point and, hopefully, you will be able to exit it through the single exit point. While in the maze, you don't see much, but at each junction, each corridor has a label that tells you if it leads back towards the start, or towards the exit. The problem is: if five corridors meet at a junction, and three of them have an arrow indicating they lead towards the exit, you will have to pick one at random. The corridors differ in length, some are uphill and some are downhill, some are straight and some are not, so the maze creators estimated the time it takes an average person to pass through each corridor. Based on the map of the maze, with all the corridors and, for each, the time it takes to pass through it, the State Fair officials want to calculate the longest time it would take an average person to reach exit from the start of the maze, assuming that at each junction they follow the labels and choose one of the corridors that lead towards the exit.

Map of the maze



Path with longest passage time (6)



Input

First line will contain a single integer number T, the number of test cases ($1<=T<=100$). Each test case starts with a line containing two integers, n ($2<=n<=100$), the number of corridor junctions, and m ($1<=m<=10000$), the number of corridors. Then, m lines follow, each describing one corridor, and consisting of three integers u v w, separated by single spaces: u is the starting junction ID, v is the end junction ID, and w is the corridor passage time (w is in 1,...,100) range. Junction IDs u,v are integer in [0,...,n-1] range, and it is also guaranteed that $u<v$. Furthermore, the corridor is supposed to be traversed only in the direction from start to end junction, from u to v (that is what the arrows at corridor ends will indicate). The start of the maze is denoted as junction 0, and the exit from the maze is denoted as junction n-1.

Output

Print T lines, one per test case. On each line, print one integer, the longest possible sum of corridor passage times starting from junction 0 (maze entrance) to junction n-1 (maze exit).

Sample Input

```
1
4 5
0 1 2
0 2 1
1 2 3
1 3 1
2 3 1
```

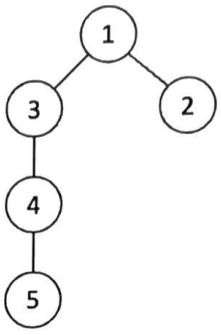
Sample Output

```
6
```

Problem I
Ants

The ants go marching, one by one... but this time, they got lost. They found themselves stuck at a branching point on a big tree, and don't know where their nest is. So, they decided that they will all go in different directions, so that each branching point, and each tip of each tree branch will be visited by at least one ant, and maybe their nest will be visible from one of these points. That will be a lot of marching; can you calculate how much? Given an unweighted tree consisting of n vertices, and a starting vertex u in the tree, compute the sum of distances along the tree (smallest number of edges that need to be traversed) from that vertex u to all the other vertices in the tree.

For example, for the tree below,



if the starting vertex is u=3, then the distances d(u,v) are: d(3,1)=1 , d(3,2)=2 , d(3,4)=1, d(3,5)=2, and the answer is 6. If u=5, then the answer is 1+2+3+4 = 10.

Input

First line will contain a single integer number n, (2<=n<=100), the number of vertices in the tree. Then, n-1 lines will follow, each describing one edge in the tree, by using two numbers, the vertices at the ends of the edge. The vertices are numbered 1 to n. The last line of input will consist of a single integer from 1,...,n range, the starting vertex.

Output

Print n lines.

Sample Input

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

Sample Output

```
6
```

Problem J
Ingredients

Alice, while exploring the Wonderland, came across a library and has stumbled upon an ancient, dusty book in the corner. This book contains the secret to a long-forgotten magic potion that promises extraordinary powers to those who drink it. However, the recipe is encrypted in a simple numerical code. Alice needs to decipher the code to unlock the secrets of the potion. The code involves two numbers: the total amount of two magical ingredients needed and the difference between those two ingredients. The Alice's task is to find out exactly how much of each ingredient is required. Specifically, given the sum and the difference of two integers, she needs to figure out that the two integers are. It is guaranteed that the two integers will be positive. The difference is also guaranteed to be positive, that is, the first unknown integer will be larger than the second.

Input

First line will contain a single integer number *S (2<=S<=1000)*, the sum of the two unknown integers. Second line will contain a single integer number *D* (1<=*D*<=998), the difference of the two unknown integers.

Output

Print two lines. The first line should contain the first, larger integer from the solution, the second line should contain the second, smaller, integer.

Sample Input

```
35
5
```

Sample Output

```
20
15
```