Gru has not been in the limelight for a long time and is, therefore, planning something particularly nefarious. Frustrated by his minions' incapability which has kept him away from the limelight, he has built a transmogrifier — a machine which mutates minions.

Each minion has an intrinsic characteristic value (similar to our DNA), which is an integer. The transmogrifier adds an integer **K** to each of the minions' characteristic value.

Gru knows that if the new characteristic value of a minion is divisible by 7, then it will have Wolverine-like mutations.

Given the initial characteristic integers of **N** minions, all of which are then transmogrified, find out how many of them become Wolverine-like.

Input Format:

The first line contains one integer, **T**, which is the number of test cases. Each test case is then described in two lines.

The first line contains two integers **N** and **K**, as described in the statement.

The next line contains **N** integers, which denote the initial characteristic values for the minions.

Output Format:

For each testcase, output one integer in a new line, which is the number of Wolverine-like minions after the transmogrification.

Constraints:

- **$1 \leq T \leq 100$**
- **$1 \leq N \leq 100$**
- **$1 \leq K \leq 100$**
- All initial characteristic values lie between 1 and 10^5, both inclusive.

Example

Input:

```
1
5 10
2 4 1 35 1
```

Output:

```
1
```

Explanation:

After transmogrification, the characteristic values become {12,14,11,45,11}, out of which only 14 is divisible by 7. So only the second minion becomes Wolverine-like.

Petr is organizing Petr Mitrichev Contest #11. The top **N** coders according to codechef ratings (excluding Petr himself) agreed to participate in the contest. The participants have been ranked from **0** to **N-1** according to their ratings. Petr had asked each participant to choose a coder with rating higher than himself/ herself, whom he/she would like to be team-mates with, and the $\mathbf{i}^{th}$ ranked coder's choice is stored as **choice[i]**. As expected, programmers turned out to be lazy, and only a few of them actually filled in their choices. If **choice[i]** = -1, then it means the $\mathbf{i}^{th}$ coder was lazy and did not fill his/her choice. Of course, the top-ranked person (i.e., rank = 0), Gennady Korotkevich (obviously), despite clearly not being a lazy person, was not able to fill any choice, because he is the top ranked coder, so **choice[0]** is always equal to -1.

Petr, being the organizer, had to undertake the arduous task of filling in the choices for the lazy participants. Unfortunately, he also got lazy and adopted the following random strategy:

For each lazy person **i** (i.e., for all **i** > 0, such that **choice[i]** = -1), he flips an unbiased coin (i.e. with 1/2 probability it lands Heads, and with 1/2 probability it lands Tails). Notice that no coin is flipped for **i**=0, since Gennady is not lazy despite his choice being -1.

If it lands Heads, he will not change the choice of this person. That is, he leaves **choice[i]** as -1.

Otherwise, if it lands Tails, he will uniformly at random select one of the top **i** ranked participants as choice of $\mathbf{i}^{th}$ person. That is, he sets **choice[i] = j**, where **j** is randomly and uniformly chosen from **0** to **i-1**, inclusive.

After this process of filling the **choice** array, Petr is wondering about the maximum number of teams he can have such that all the valid choices are respected (i.e. if **choice[i] = j** and **j** != -1, then **i** and **j** should be in the same team). Petr now has to arrange for computers. As each team will need a computer, he wants to know the expected value of maximum number of teams. As he is too busy in organizing this contest, can you please help him?

Input:

The first line of input contains **T**, the number of test cases. Each test case contains 2 lines.

The first line contains a single integer **N**.

The second line contains **N** integers, which specify the choice array: **choice[0], choice[1],..,choice[N-1]**.

Output:

Each test case's output must be in a new line and must be 1 number which is the expected number of teams. Your answer should be within an absolute error of 10^{-6} from the correct answer.

Constraints

- **1 ≤ T ≤ 10**
- **1 ≤ N ≤ 1000**
- The sum of all the **T** **N**s in one test file ≤ 5000
- **-1 ≤ choice[i] ≤ i-1** , for all **0 ≤ i < N**
- **choice[0]** will be equal to -1, always

Example

```
Input:
1
5
-1 -1 1 2 1
```

Output:
1.5

Explanation:

choice[2] = **1** implies **1** and **2** should be in the same team. **choice[3]** = **2** implies that **2** and **3** have to be in the same team. Therefore, **1**, **2** and **3** have to be in the same team. Also, **choice[4]** = **1** implies that **1** and **4** have to be in the same team, which means that **1, 2, 3** and **4** — all have to be in the same team.

The only lazy person is **1**. So with probability 1/2, Petr leaves his choice unchanged, or with the remaining 1/2 probability, he assigns **choice[1] = 0**, as **0** is the only higher ranked person for **1**. In the first case, you can get 2 teams: {**0**} and {**1, 2, 3 ,4**}.

In the second case, because **choice[1] = 0**, **0** and **1** are forced to be in the same team, and hence {**0, 1, 2, 3, 4**} is the only possible team. Therefore, with 1/2 probability, number of teams = 1 and with another 1/2 probability, it is 2. Hence, expected number of teams = (1/2 * 1) + (1/2 * 2) = 1.5.

Jimma the cat had placed **N** bowls of cat food in a line. The amount of food in each bowl is an integer and is given to you as the array **A[1], A[2], ... , A[N]**, where **A[1]** is the amount of food in the leftmost bowl, and **A[N]** is the amount in the rightmost bowl.

But Chingam the kitten doesn't like unsorted sequences. In general, for an arbitrary sequence **B[1], B[2], ... , B[N]**, Chingam's **unhappiness** is defined as the number of inversions in that sequence. That is, **unhappiness** = number of pairs **(i,j)**, such that **i < j** and **B[i] > B[j]**.

Now, Chingam wants to take the bowls from the old line and create a new line with them, possibly re-ordered, so that his **unhappiness** is minimized. ie. let's say that after re-ordering, the bowls form the sequence **P[1], P[2], ... , P[N]**. That is, **P[1]** is the amount of food in the leftmost bowl of the new line and **P[N]** is the amount of food in the rightmost bowl of the new line. Then he wants to minimize his **unhappiness** with respect to this array P.
Note that **P[1], P[2], ... , P[N]** will be a permutation of **A[1], A[2], ... , A[N]**.

But he follows a peculiar method to build the new line (which is initially empty) from the old line of bowls. He takes one bowl at a time from the old line and places it in the new line. Due to the limitations of his paws, Chingam can only pick a bowl which is the current leftmost or rightmost in the old line for each step. And he can place it only at the current right or left end of the new line.

By doing this, Chingam gets a new sequence **P[1], P[2], ... , P[N]**. He wants to minimize his **unhappiness** with respect to this new sequence.

Find out the minimum **unhappiness** that Chingam has to bear.

Input Format:

The first line contains one integer, **T**, the number of test cases.

For each test case:

The first line contains a single integer, **N**.

The next line contains **N** integers: **A[1], A[2], ... , A[N]**.

Output Format:

Output one integer in a new line for each testcase: the minimum **unhappiness** achievable.

Constraints:

- **1 ≤ T ≤ 10**
- **1 ≤ N ≤ 100**
- **0 ≤ A[i] ≤ 1000**, for all **i** in **1** to **N**.

Example:

Input:
```
1
6
5 3 4 3 2 7
```

Output:
```
1
```

Explanation:

We show a sequence of moves which gives **unhappiness** = 1:

```
Old Line = 5 3 4 3 2 7
New Line = Empty

Old Line = 3 4 3 2 7
New Line = 5

Old Line = 3 4 3 2
New Line = 5 7

Old Line = 4 3 2
New Line = 3 5 7

Old Line = 3 2
New Line = 4 3 5 7

Old Line = 2
New Line = 3 4 3 5 7

Old Line = Empty
New Line = 2 3 4 3 5 7
```

Now, in this new line, **i = 3** and **j = 4** is the only pair which satisfies **i < j** and **P[i] > P[j]**. Therefore, the **unhappiness** of this sequence is 1. You can verify that this is the minimum possible.

You are given **N** lines (which extend indefinitely in both directions) on the **X-Y** plane. Some of these lines intersect with each other and others don't (parallel lines). We call each such intersection, an intersection point. You have to find the $\mathbf{K}^{th}$ intersection from the bottom.

In particular, you have to output the smallest value **V**, such that there are at least **K** intersection points whose **Y**-coordinate is $\leq$ **V**.

If **m** lines intersect at a particular point, that point is counted **m(m-1)/2** times. That is, if only two lines intersect at a point, it is counted as one intersection point. But if three lines intersect at the same point, it is counted as 3 intersection points: one for each pair.

You are guaranteed that all the **N** lines are distinct, and that there are at least **K** intersection points between the lines.

Input Format:

The first line contains the two integers **N** and **K**.

The next **N** lines contain two space separated integers each. The $\mathbf{(i+1)}^{th}$ line contains $\mathbf{M_i}$ and $\mathbf{C_i}$, such that the equation $\mathbf{y = M_i*x + C_i}$, denotes the $\mathbf{i}^{th}$ line.

Output Format:

Output one value denoting **V** as defined in the problem, which should be within 10^{-2} of the correct answer.

Constraints:

- $\mathbf{1 \leq N \leq 10^5}$
- $\mathbf{K} \geq 1$
- It is guaranteed that there are at least **K** intersection points. That is, $1 \leq \mathbf{K} \leq$ number of intersection points in the input data
- $\mathbf{-20000 \leq M_i \leq 20000}$ and $\mathbf{M_i} \neq 0$
- $\mathbf{-20000 \leq C_i \leq 20000}$
- It is guaranteed that there are no **i** and **j**, such that $\mathbf{i \neq j}$ and ($\mathbf{M_i = M_j}$) and ($\mathbf{C_i = C_j}$)

Example:

Input 1:
```
4 6
-1 2
-3 -1
3 1
-4 1
```

Output 1:
```
3.5
```

Explanation:

There are 6 intersection points in total. Therefore, we want the **y**-coordinate of the highest intersection point. We see that the highest intersection point is (-1.5,3.5), and hence the answer is 3.5.

Input 2:
```
4 3
-10 -6
-5 -10
```

2 1
4 -8

Output 2:
-7.429

A tree is one of the most beautiful data structures. Given a tree of **N** nodes (numbered **1** to **N**), we can augment it by connecting any pair of nodes which sharing one or more neighbour/s. In other words, the augmentation operation joins any nodes of tree within distance two with an edge. The resulting graph is called the augmented graph for the given tree. Due to having additional edges, the total number of edges in the augmented graph is **M**, instead of **N-1**.

Fortunately, **M** is not very large, and therefore, it is possible to reconstruct the tree given its augmented graph! In this problem, you are going to read the augmented graph and you have to find the original tree for that. Note that there might be different trees leading to the same augmented graph. **You may output any one of them.**

Input

The first line contains an integer **T** denoting the total number of test cases.

In each test case, the first line contains two integers **N** and **M** denoting the number of nodes and edges in the augmented graph, respectively.

This is followed by **M** lines containing two space-separated integers **a** and **b** denoting that there is an edge between vertex **a** and **b**.

Note that it is always guaranteed that the input graph is always the augmented graph of some tree.

Output

For each test case, output **N-1** lines, each of which contains two space-separated integers indicating one edge in the original tree.

Constraints

- $1 \leq T \leq 10$
- $1 \leq N \leq 10^4$
- $0 \leq M \leq 10^5$
- $1 \leq a, b \leq N$
- Augmented graph does not contain loop or multiple edges.

Example

Input:

```
2
4 5
1 2
1 3
3 4
2 3
1 4
5 7
1 2
2 3
3 4
4 5
1 3
2 4
3 5
```

Output:

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

2 3
3 4
4 5

You are in a magically discrete 2D world: an **H * W** grid containing **H** rows and **W** columns of cells. There are rocks at various positions **(x, y)**, none of which can be at **(1, 1)** at the beginning, but there may be rocks in any other cell of the grid.

(1, 1) refers to the bottom-left cell, and **(W, H)** refers to the top-right cell.

The most interesting fact in this magic 2D world is that the rocks are falling down without any acceleration! More specifically, a rock which is at **(x, y)** at time **t** will be at **(x, y - 1)** at time **t+1** and it will disappear once **y** becomes **0**.

At time **t = 0**, you are at the bottom-left cell of the grid denoted as **(1, 1)**. (Note that this means that there is **no rock at the starting point (1, 1)** at the beginning.) You need to move left or right or stay at the same position in order to avoid being hit by a rock. That is, if you are at **(x, 1)** at time **t**, you could be at any of the cells **(x - 1, 1), (x, 1), or (x + 1, 1)** at time **t + 1**, assuming that cell exists.

Along with space, time is also discrete in this magic world. In other words, this world is quantum physics gone berserk! At every second, you make your move, and then the rocks fall 1 cell down. So you should make sure that when you move to a cell at time **(t + 1)**, there is no rock in that cell at **time t as well as (t + 1)**.

We are interested in computing how many different layouts of rocks you can survive if you always play in the optimal way after the layout is made available to you. The answer might be very large, so you only need to output answer modulo **M**.

Input

The first line contains an integer **T** denoting the total number of test cases.

For each test case, there are three single-space separated positive integers **W**, **H**, and **M**, in a single line.

Output

For each test case, output a line with the answer.

Constraints

- $1 \leq T \leq 10$
- $1 \leq W \leq 7$
- $1 \leq H \leq 100$
- $1 \leq M \leq 10^9 + 7$

Example

Input:

```
2
2 2 100
2 3 100
```

Output:

```
5
12
```

Explanation

When **W=2, H=3**, we can discuss the problem in three cases as following, where **Y** stands for you, **E** stands for an empty cell, **R** stands for a rock, and **?** stands for **either E or R**.

(1) 8 layouts in total.

```
E?
E?
Y?
```

In this case, you just stay at **(1, 1)** throughout.

(2) 2 layouts in total.

```
? E
RE
YE
```

In this case, you move to **(1, 2)** at the very beginning, and stay there.

(3) 2 layouts in total.

```
RE
EE
Y?
```

In this case, you stay at **(1, 1)** for one second, and then move to **(2, 1)**.

So, we have a total of 12 layouts which we can survive.