Problem S: Solitaire

Maggie has a special deck of cards. There are $4n$ cards in the deck: For each i between 1 and n, inclusive, there are four cards with the value i – the i of spades, the i of hearts, the i of diamonds, and the i of clubs. Maggie uses this deck of cards to play a simple solitaire game.

At the beginning of the game the deck of cards is placed face down on the table. There are four empty slots – one for each suit. Additionally, there is an empty discard pile next to the deck. The goal of the game is to place all cards of each suit onto the corresponding slot. The cards have to be placed onto their slot in ascending order. A more precise description of the game follows.

During the game, Maggie repeats the following steps:

1. Pick up the topmost card of the deck.
2. Look at its suit and look at the slot for the corresponding suit.
3. If the card is an ace (i.e., its value is 1), place it onto the corresponding empty slot.
4. If the value of her card is $v > 1$ and the topmost card on the corresponding slot has value $v - 1$, place the card with value v on top of the card with value $v - 1$. For example, the 8 of spades can be placed onto the 7 of spades.
5. If you were unable to place the current card onto its slot, place it face up onto the discard pile. For example, if your current card is the 8 of spades and the top card on the corresponding slot is the 3 of spades, the 8 of spades goes onto the discard pile.
6. If there are no cards left in your deck:

   If there are no cards left in your discard pile (i.e., all $4n$ cards have been placed onto their slots), the game ends.

   Otherwise, take the discard pile and flip it upside down to produce a new deck. Note that the order of cards in the new deck is the same as the relative order of these cards in the original deck.

Problem specification

Obviously, Maggie will always win the game eventually, but sometimes it can take quite long.

You are given n and the initial order of cards in the deck. Compute how many times Maggie went through her deck of cards before winning the game. In other words, the answer you should compute is one plus the number of times she flipped the discard pile to get a new deck.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of two lines. The first line contains the number n. The second line contains a list of $4n$ different cards – the initial deck from top to bottom. Each card is given in the format `Xy`, where `X` is one of the four suits and `y` is the value. E.g., `H47` is the 47 of hearts.

In the **easy subproblem S1** we have $t = 20$ and $n = 13$ in each test case. (Maggie is playing with the standard deck of 52 cards.)

In the **hard subproblem S2** we have $t = 12$ and $1 \leq n \leq 150,000$ in each test case. Because the input file size for subproblem S2 is about 30 MB, you cannot download it directly. Instead, we have provided a small Python 2 program that will generate the file `s2.in` when executed.

Output specification

For each test case output a single line with a single integer: the number of times Maggie has to go through her deck in order to win the game.

Note

In the real contest some large inputs will be provided in the same way as the input `s2.in` in this practice problem. Please make sure you are able to generate it.

Example

input

```
1

3
S2 H3 S1 D3 D1 H1 D2 C1 S3 H2 C2 C3
```

output

```
2
```

After going through her deck once, the topmost cards on the four slots will be the 1 of spades, the 2 of hearts, the 2 of diamonds, and the 3 of clubs. The remaining cards will now be in the discard pile. Afterwards, Maggie will flip the discard pile over and she will go through the remaining cards again (starting again with the 2 of spades). During this second pass Maggie will be able to place all the remaining cards.

Task authors

Problemsetter:	Michaela 'Šandyna' Šandalová
Task preparation:	Michaela 'Šandyna' Šandalová
Quality assurance:	Michal 'mišof' Forišek

Solution

The easy subtask can easily be solved by a straightforward simulation of the game.

In the worst possible case, the simulation will take $\Theta(n^2)$ steps. The worst possible case is actually pretty simple: a sequence of cards of the same suit, sorted in descending order. In each pass through the deck you will only be able to place one of the cards onto their slot. A straightforward simulation for $n = 250\,000$ would take hours. Given enough computational power, and considering the length of our practice session, it was possible to solve the hard subproblem in time – but there clearly has to be a better way!

One thing that should be obvious is that we can consider each suit separately. That is, we can separately compute the number of rounds we need to play each suit, and then compute the final answer as the maximum of those four numbers.

Hence, we just need to come up with a good way to solve the problem for a single suit. Let's discover it on an example. Assume that the spades in our deck are ordered as follows:

`S10 S8 S1 S4 S7 S6 S5 S11 S2 S13 S9 S3 S12`

What will happen in the first pass through the deck? We discard the 10, discard the 8, put the 1 onto the slot for spades, discard a bunch of other cards, put the 2 onto the 1, discard the 13, discard the 9, put the 3 onto the 2, and discard the 12. That's it for the first pass.

You may now note that we managed to remove the first three cards from our deck: first the 1, then the 2, and then the 3. We were able to remove the 2 in the same pass as the 1 because the 2 appeared later in the deck. We were able to remove the 3 in the same pass as the 2 because the 3 appeared later in the deck as the 2. And we were unable to remove the 4 in the same pass, because by the time we got to the 3 we already discarded the 4.

But that's precisely the observation we needed! For each i, a new round will begin between cards i and $i+1$ are played if and only if card $i+1$ appears before card i in the deck.

Using this observation, we can easily compute the number of rounds in linear time: First, we go through the deck and for each value we note its index in the deck. Then, we go through all values and for each value we compare its index to the index of the next value.

Problem T: Town

You are standing in a town with infinitely many houses. Currently, the houses do not have any house numbers. You were given the task to fix this.

You have a box with plastic digits. For each i between 0 and 9, inclusive, there are d_i copies of the digit i in your box. You can number a house by sticking the appropriate digits to its wall. For example, on the house number 474 you will use two digits 4 and one digit 7.

You have decided that you will number the houses sequentially, starting from 1. How many houses can you number before you run out of digits?

Problem specification

You are given the counts $d_0, \dots, d_9$ of the digits in your box. Find the largest x such that you are able to write the numbers 1 through x using your set of digits.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case consists of a single line containing 10 nonnegative integers – the counts of digits 0 through 9.

In the **easy subproblem T1** the sum of all d_i will be at most 10^7.

In the **hard subproblem T2** the sum of all d_i will be at most 10^{16}.

Output specification

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

Example

input

```
1

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

output

```
10
```

With the digits you have, you are able to build the numbers 1 through 10. Once you do so, you will be left with three digits: one 1, one 4, and one 7. This is not enough to construct the number 11.

Task authors

Problemsetter:	Monika Steinová
Task preparation:	Monika Steinová
Quality assurance:	Michal 'Mimino' Danilák

Solution

The easy subtask can be solved by brute force: generate house numbers starting from 1, for each of them count the digits used and terminate once you don't have enough digits to build the current number.

For the hard subtask this solution would be too slow. How can we improve it? The key is to notice that the answer is monotonous: if we have enough digits for houses 1 through x, we have enough digits for houses 1 through y, for any $y < x$. This makes it possible to determine the optimal answer using binary search.

In order to perform the binary search we have to implement a function that takes an n and verifies whether the box contains enough digits to produce numbers from 1 to n. This function will somehow count the digits used in the numbers from 1 to n and compare those counts to the counts of digits in our box.

We will now show how to calculate the value $C_{d,n}$: the number of times the digit d occurs in the numbers 0 through $n-1$. The value $C_{d,n}$ can easily be computed recursively in $O(\log n)$ time as follows:

- If $n = 0$ then $C_{d,0} = 0$.
- If $n \bmod 10 \neq 0$, then $C_{d,n}$ is $C_{d,n-1}$ + the number of times d occurs in $n-1$.
- If $n \bmod 10 = 0$, imagine that you listed all the numbers from 0 to $n-1$.
  Look at the last digit. It cycles through 0-9 exactly $n/10$ times, hence you have $n/10$ copies of digit d there.
  Now erase the last digit. What is left? The sequence of numbers 1 through $(n/10) - 1$, each repeated 10 times. How many copies of your digit d are there? Obviously, for $d \neq 0$ the answer is $10 \cdot C(d, n/10)$.
  There is a special case with $d = 0$: the first 10 numbers in our list (0-9) only had a single digit each. After erasing those nothing was left (as opposed to ten 0s). Hence, for $d = 0$ we have to subtract those from the result: we only have $10 \cdot (C(0, n/10) - 1)$ zeros among the digits other than the last digit.

Once we know how to compute the values $C_{d,n}$, we are all set. During the binary search we will compute the value $C_{0,n+1} - 1$ and the values $C_{1,n+1}$ through $C_{9,n+1}$ and we will compare these to the ten values we were given as the input.

Problem U: Unusual Game Show

You may have already heard about the famous game show host Monty Hall. Back in the day, his game show had confused many a bright mathematician.

This is how it all looked like: The contestant was shown three doors, labeled 1 through 3. There was a prize (e.g., a new car) behind one of the doors, and a goat behind each of the other two doors. The door hiding the prize was chosen uniformly at random. Monty knew which door contains the prize. The game consisted of three steps:

1. At the beginning of the game, the contestant was asked to choose one of the three doors for herself.
2. Once the choice was made, Monty would open one of the doors the contestant did not choose.

   Of course, Monty would never open the door with the prize. If the contestant chose the door with the prize, Monty would open either of the other two doors, chosen uniformly at random. In all other cases Monty would open the only door that was neither chosen by the contestant nor hiding the prize.
3. Then, Monty asked the contestant a very tricky question: "Do you want to *keep* the door you have, or do you want to *switch* to the other door?"

   Once the contestant made her final decision, Monty opened the door she chose to show whether she found the prize.

This game became very famous among mathematicians because the optimal strategy is very counter-intuitive. At the end of the game, the player gets to choose between two doors. One of them contains the prize, the other does not. Thus, on the surface it seems that the choice doesn't matter and that the probability of winning the prize is always 1/2. **This is not true.** It can be shown that the optimal strategy for the contestant is to *never keep, always switch to the other door.* With this strategy, the actual probability of winning the prize is 2/3.

Monty's game today

Monty Hall is still hosting the game show. However, there have been some changes:

- For financial reasons, the number of doors is now d ($d \geq 3$). There is one prize and $d - 1$ goats.
- Monty is very old. When performing a show, with probability p he is tired.

  If Monty isn't tired, he follows the above protocol. However, if he is tired, he wants to avoid unnecessary walking. Therefore, if he has a choice in step 2, he will always open **the door with the smallest number** among the doors he is allowed to open. (He is still not allowed to open the door with the prize nor the door currently chosen by the contestant.)
- In step 3, the contestant gets to choose whether she keeps the door she has or switches to *any other unopened door*. When switching, the contestant gets to choose which one of the other doors she now wants.
- After step 3, the game is over. The remaining $d - 1$ doors are opened and it is revealed whether the contestant won the prize.

Problem specification

You are given the number of doors d and the probability p. Find an optimal strategy for the contestant, and report her probability of winning the prize if she follows that strategy.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case consists of a single line containing the numbers d and p. The value p is always a number from $[0, 1]$ with exactly 6 decimal places.

In the **easy subproblem U1** we have $t = 10$ and $d = 3$.

In the **hard subproblem U2** we have $t = 100$ and $3 \leq d \leq 100$.

Output specification

For each test case, output a single line with a single real number: the optimal probability of winning the prize. Output at least 10 decimal places. Answers within 10^{-9} of our answer will be accepted as correct.

Example

input

```
1

3 0.000000
```

output

```
0.666666666666666667
```

In this example we have 3 doors and Monty is never tired, so we are playing the original game.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Monika Steinová

Solution

The easy subtask could be solved really easily. The key is to realize that the change in rules doesn't actually change anything. Regardless of which door Monty opens, he is essentially telling the contestant: *"If your initial choice was wrong, this other unopened door is where the prize is."* Hence, *always switch* is still the optimal strategy, and the win probability is 2/3: you lose if and only if your initial door choice was correct.

In the general case this is no longer true, Monty's tiredness can be a useful source of information. For example, imagine that Monty is always tired and there are 10 doors. If you start by choosing door 1 and then Monty opens door 3, it is certain that the prize is behind door 1 or door 2 (and not any of the doors 4 through 10).

Luckily, the optimal strategy can still be found easily. We can simply compute all the probabilities and use them to make the optimal decision in any state of the game.

We are interested in the following conditional probabilities: *Given that I chose door a and then Monty opened door b, what is the probability that the prize is behind door c?*

We can compute these probabilities directly from their definition: as the actual probability of that event, divided by the probability that Monty will open door b in any situation. And the denominator is simply a sum over whether Monty is tired or not, and over all doors that can contain the prize.

Once we know the conditional probabilities described above, the optimal strategy follows. First, for a fixed door a we can, for each b, evaluate the probability that Monty opens b and if he does, we find the door c that is most likely to contain the prize and switch to that door. In this way, we can compute our probability of winning if we start by choosing door a. Finally, we do that for each possible a and pick the best one. See our sample implementation for more details.

Of course, instead of letting a program solve the game for us we can also solve it on paper. Let's see what we can derive about it.

First of all, the initial door choice still doesn't actually matter. The situation after our initial choice always looks the same: there is one chosen door and an ordered sequence of doors that weren't chosen. Thus, without loss of generality, we will assume that we chose door d.

There are now three possibilities for what Monty will do:

- He may open door 1. Regardless of whether he did so because he was tired or because he chose to do so at random, the only information we get is that door 1 doesn't contain the prize. All other doors (2 through $d-1$) are equally likely to contain the prize, and that probability is now greater than $1/d$ so we should switch to any of them.
- He may open door with a number greater than 2. This means that the door had to be chosen at random, and again we know nothing about the other doors. Thus, again the optimal strategy is to switch to any other door.

- He may open door number 2. This is the interesting part, because it may be one of two distinct cases: either he chose it at random, or he is tired and he was forced to choose it because door 1 contains the prize. If the second case has a nonzero probability, it skews the probability in favor of door 1. Hence, the optimal strategy in this case is to switch to door 1.

We may therefore summarize one optimal strategy as follows: **Start by choosing door d. Once Monty opens some other door, switch to the unopened door with the smallest number.**

And from this summary we can easily compute our win probability: we win if either the prize is in door 1 (probability $1/d$) or if the prize is in door 2 and Monty opens door 1 (probability 1 if he is tired and $1/(d-2)$ if he is not).

Thus, the final answer is

$$\frac{1}{d} + \frac{p}{d} + \frac{1-p}{d(d-2)}$$