IPSC 2015

problems and sample solutions

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 $d_0, \ldots, d_9$ 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 2 (and not door 1, 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

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

Problem A: A+B

Our first problem today is as easy as $a + b$.

Problem specification

You are given a string of digits. Rearrange those digits to build two nonnegative integers a and b such that the sum $a + b$ is as large as possible.

Each number must consist of at least one digit. Leading zeros are not allowed, but the number zero consisting of a single digit 0 is allowed. You have to use each digit exactly as many times as it occurs in the given input string.

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 one string of digits. If there are more than 2 digits in the string, not all of them are zeros.

In the **easy subproblem A1** we have $t = 900$ and each test case consists of exactly 3 digits.

In the **hard subproblem A2** we have $t = 1000$ and the number of digits in each test case is between 2 and 16, inclusive.

Output specification

For each test case, output a single line with a single integer: the largest sum that can be achieved.

Example

input

```
4

001

175

21

444444
```

output

```
10
76
3
44448
```

The first two test cases could appear in the easy input file `a1.in`*, the other two could only appear in the hard input file* `a2.in`*.*

One optimal arrangement of digits for each example test case: 10+0, 71+5, 1+2, and 44444+4.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'Žaba' Anderle
Quality assurance:	Jano Hozza

Solution

Our task is to rearrange the string of digits into two integers a and b such that $a + b$ is as large as possible. Let's see how we can construct such a and b, and thus find the value of $a + b$.

In the easy subproblem, we had exactly three digits. The answer will certainly be of the form $\overline{ab} + \overline{c}$, where a, b and c are the three given digits (in some order), and the overline denotes a number that consists of the given digits.

The value of the above number is $10a + b + c$. From this it is obvious that a should be the largest of the three given digits and that the order of b and c does not matter.

The above observation can easily be generalized to solve the hard subproblem as well. The optimal solution is to read the string of digits, sort them in non-ascending order, and then take the first $n - 1$ digits as one of the numbers and the last digit as the other number. For example, the optimal solution for the input `12345` is `5432+1`.

Formal proof

It is easy to verify that the solution described above will never run into troubles with unnecessary leading zeros – if there are zeros in the input, one of them will be b (which is valid) and all others will be at the end of a (which is also valid).

Lemma: In the optimal solution one of the two numbers will consist of just a single digit.

Proof: Consider an arbitrary arrangement of digits. Label the two numbers a and b so that $a > b$. Suppose that b has more than one digit. Let's now take the last digit of b and append it to a instead. It should be obvious that we didn't create any new leading zeros anywhere, so the solution remains valid.

How will the sum change? Let's write the original b as $10b' + d$. The sum before we moved the digit was $a + 10b' + d$. After the move the new sum is $10a + d + b'$. The change is $9(a - b') > 0$, therefore the new arrangement is better than the old one, so the old arrangement cannot be optimal.

Theorem: The arrangement described in our solution above is optimal.

Proof: From the lemma we know that each optimal solution has the form $\overline{d_1 \dots d_{n-1}} + \overline{d_n}$. The value of this number is $10^{n-2}d_1 + 10^{n-3}d_2 + \dots + 10d_{n-2} + d_{n-1} + d_n$.

Hence, it is clearly optimal to assign the largest digit to d_1, the second largest to d_2, and so on. The order of the last two digits does not matter.

Problem B: Bawdy Boil-brained Bugbear

A certain gentleman shamelessly ate another gentleman's cake. This inexcusable act led to a long merciless battle between them. Of course, proper gentlemen never fight with their fists – they use carefully chosen insults built of words from Shakespeare's time. This is how the battle went on:

"Thou atest my cake, thou bawdy doghearted nut-hook!"
"Aye? But thou art a goatish half-faced bugbear."
"Fain I am not a goatish doghearted puttock as thou."
"Thou art a bawdy half-faced nut-hook."
"Thou bawdy doghearted bladder!"
"Thou goatish half-faced puttock!"
"Thou spongy boil-brained bladder!"
"Thou bawdy boil-brained bugbear!"
"Fie, thou winnest!"

Problem specification

A valid insult consists of three words: The first one must be a *simple insulting adjective*, the second one must be a *compound insulting adjective*, and the third one must be an *insulting noun*. For each of these categories we give you a list of available words. Every word has a certain strength; the strength of an insult is the sum of the strengths of its three words.

You are then given a list of insults. For each one of them you have to come up with an appropriate response – an insult which is by 1 unit stronger than the challenging insult. You are not allowed to use the same response more than once.

Input specification

The input consists of four parts. The first three parts describe the three wordlists that the insults are built from. They have the same structure: The first line contains an integer m denoting the number of words in the wordlist. Then m lines follow; on each line there is a word and its strength (a positive integer). Words consist of lowercase English letters and at most one hyphen (`-`). Each word is between 1 and 15 characters long (inclusive). The words in all three wordlists are distinct.

The last part of the input describes a list of insults. On the first line there is an integer n denoting the number of insults. On each of the following n lines there are three space-separated words. You are guaranteed that the insults are valid with respect to the former wordlists.

In the **easy subproblem B1** we have $m = 50$ and $n = 500$. Maximum word strength is 10^3.

In the **hard subproblem B2** we have $m = 10\,000$ and $n = 10\,000$. Maximum word strength is 10^6.

Output specification

Output n lines. For each i, the i-th line should contain an insult whose strength is exactly one greater than the strength of the i-th insult in the input. You cannot use the same insult more than once. (I.e., your output must contain n distinct lines.)

You are guaranteed that there is a sufficient amount of appropriate insults.

Example

input

```
3
bawdy 6
goatish 4
spongy 1
3
boil-brained 10
half-faced 6
doghearted 3
3
bladder 7
bugbear 3
puttock 7
7
spongy boil-brained bladder
goatish half-faced bugbear
bawdy half-faced bugbear
spongy doghearted puttock
goatish half-faced bugbear
goatish half-faced bugbear
bawdy doghearted bladder
```

output

```
bawdy boil-brained bugbear
goatish doghearted bladder
bawdy doghearted puttock
bawdy doghearted bugbear
spongy half-faced puttock
spongy boil-brained bugbear
goatish half-faced bladder
```

The strengths of the input insults are 18, 13, 15, 11, 13, 13, and 16.

The strengths of the output insults are 19, 14, 16, 12, 14, 14, and 17. Note that we had to use three different insults with strength 14 each.

The conversation in the beginning of this task contains a sequence of valid insults if we consider the wordlists from this sample input and an additional insulting noun 'nut-hook' of strength 3. Each insult is an appropriate response to the previous one. There is no valid response to the last insult of strength 19.

Task authors

Problemsetter:	Tomáš 'Tomi' Belan
Task preparation:	Jano Hozza
Quality assurance:	Peter 'Bob' Fulla

Solution

The easy subproblem is so small that we can solve it completely by brute force: for each input, iterate over all possible insults until you find one with a suitable strength.

This approach would, obviously, be too slow for the hard subproblem.

The first possible improvement would be to generate the possible results only once. Once we do that, we can sort them into buckets according to their strength, and we will be able to answer queries in constant time. Sadly, this still doesn't work: there are $10\,000^3$ possible insults, so we don't have enough time and space to generate and store all of them.

There are many ways how to solve the hard subproblem. Here are a few.

Meet in the middle

We may notice that the precomputing solution described above spends too much time on precomputation and too little time on answering queries. We can therefore use some kind of a meet-in-the-middle technique to obtain a better tradeoff.

During precomputation we will generate all pairs of words from the first two wordlists and we will store them into buckets according to their total strength.

Then, whenever we want an insult with a specific strength s, we can iterate over all words from the third wordlist, and for each word (with strength s_3) we look into the corresponding bucket (strength $s - s_3$) for the rest of the insult. If we get a insult we have already generated before, we skip it and find another.

Considering a reasonably low number of skips with same value, the time complexity is $O(m^2 + nm)$.

A heuristic approach

You may notice that there are $10\,000^3$ possible insults in hard input but only $3\,000\,000$ possible strengths. It is also easy to verify that word strengths are distributed reasonably. Therefore, there is a *lot* of possible insults of each type.

In order to speed up the naive precomputing solution we mentioned in the beginning, we can simply use random sampling. If we only keep a small-ish random sample of each wordlist, the solution becomes fast enough and it will still be able to find enough answers to the input insults. You just needed to make sure that you selected enough words close to minimum and maximum strength, because the number of valid combinations there is small.

Dynamic programming

The task can also be solved using a knapsack-like dynamic programming algorithm: for each strength we store some of the ways how we got there.

Problem C: Chess Pieces

Yes, we do have an actual chess problem this year. In both subproblems we consider a standard chess game. The game is played:

- on a standard 8×8 chessboard from the standard initial configuration
- according to all standard chess rules that matter (e.g., including castling and en passant)

Problem specification

At the beginning of a chess game there are at most 8 pieces of each type on the board. For example, there are 8 white pawns, 8 black pawns, 2 white rooks, and only 1 black queen. In the **easy subproblem C1** we want you to produce any valid sequence of chess moves that will lead to a board that contains *more than 8 pieces* of any specific type.

At the beginning of a chess game there are 4 rooks on the chessboard: two white and two black ones. In the **hard subproblem C2** we want you to produce any valid sequence of chess moves that will lead to a board that contains *the largest possible total number of rooks.*

Input specification

There is no input.

Output specification

Output a valid plain text (7-bit ASCII) file containing a sequence of alternating white's and black's halfmoves in PGN notation. Any sequence of moves will be accepted as long as the final configuration has the desired property. More technical details are given at the end of this problem statement.

Example output

```
1. a4 b5    2. axb5 Nc6    3. b6 Nf6    4. b7 Nd5    5. bxa8=N Ndb4    6. f3 g6
7. g4 Bh6    8. Bh3 Nd3+    9. Kf1 O-O    10. Kg2
```

This is not a valid solution to any subproblem. It is just an arbitrary sequence of moves. The example has valid syntax that can be correctly parsed. Note that in turn 5 the white pawn got promoted to a knight instead of a queen.

Technical details

Your submissions will be parsed with the `chess.pgn.read_game` method of the python-chess library, version 0.8.1. Hence, we expect you to submit the record of a game in a valid PGN notation. The library should be pretty tolerant – it should be able to parse anything valid, including comments, headers, NAGs and games with multiple variations (i.e., branches of play). Still, **we recommend that you just submit a plain ASCII log that is as simple as possible**. The example above probably contains all the notation you'll need to use.

In particular, if the notation of your game contains variations, we give you no guarantee about which branch we will choose to evaluate.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Vlado 'usamec' Boža
Quality assurance:	Michal 'mišof' Forišek

Solution

If you wanted to solve the easy subproblem separately, probably the most simple strategy was white's all out offense: make all white pawns reach the last row and promote them all into queens.

Here's a rather short game that does exactly that:

```
1. a4 a5 2. b4 b5 3. c4 c5 4. d4 d5 5. e4 e5 6. f4 f5 7. g4 g5 8. h4 h5 9. axb5 Kf7 10.
bxa5 Kg7 11. cxd5 Kh7 12. dxc5 Nf6 13. fxe5 Rg8 14. exf5 Kh8 15. hxg5 Nh7 16. gxh5 Be7
17. a6 Bf6 18. gxf6 Qe8 19. b6 Qg6 20. hxg6 Nd7 21. b7 Rb8 22. e6 Ne5 23. e7 Nf7 24.
gxf7 Ra8 25. f8=Q Bd7 26. e8=Q Ra7 27. b8=Q Rb7 28. a7 Rc7 29. a8=Q Rb7 30. c6 Ra7 31.
c7 Rb7 32. c8=Q Bc6 33. d6 Bb5 34. d7 Bc6 35. d8=Q Bd7 36. Qfc5 Bc6 37. Qe4 Be8 38. f7
Rc7 39. fxe8=Q Rb7 40. f6 Rc7 41. f7 Rd7 42. f8=Q
```

Note that 40 moves are necessary just to be able to get all white pawns to the last row. As you can see, our solution uses 42. Here's a bonus challenge for you: is that optimal, or can you solve it in 41, or even in 40 moves?

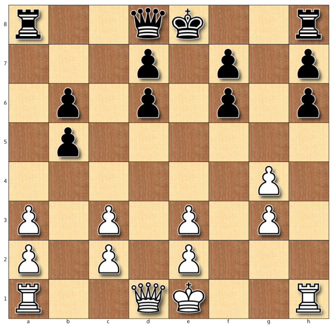
Update: Samuel Huang has e-mailed us to point out that 35 moves could be enough – if you promote just seven pawns into other pieces. And indeed, the easy subproblem can be solved in 35 moves and that is the minimum.

Hard subproblem

Of course, another approach was to realize that any game that solves the hard subproblem also solves the easy one. You could start solving the hard subproblem and then submit its log twice to solve both subproblems. You could even optimize: halfway through solving the hard subproblem, once you have 9 pieces of the same type, submit the log of the game so far, and then continue solving.

What is some theoretical upper bound on the number of rooks? Initially there are four, and there are sixteen pawns that can be promoted, so that would give us a total of twenty rooks. Of course, that's assuming that no pawn gets ever taken. At the first glance, that seems impossible. I mean, how do you even get the white pawns past the black ones *and vice versa* without some of them taking the others?

And figuring out how to do that was indeed the main challenge in this subproblem. The picture below tells the whole story. It a configuration our solution reaches at some point. From this configuration it should be fairly obvious both how we reached it and how the game will now proceed.



Problem D: Dijkstra's Nightmare

Every computer scientist should be familiar with Dijkstra's algorithm: the basic algorithm to compute single-source shortest paths in a given graph. When executing the algorithm, we maintain a set of *active* vertices. In each iteration, we select and process the active vertex with the smallest current distance. When processing a vertex, we examine the outgoing edges. Whenever one of them can be used to improve the distance to an adjacent vertex, we do so and mark that adjacent vertex as active.

It is well-known that the algorithm is efficient whenever all edge lengths are non-negative. The reference implementation we provided in this problem runs in $O(m \log n)$ time for such graphs (where m is the number of edges and n is the number of vertices).

Things start getting hairy once we allow edges with negative lengths. Finding the shortest *simple* path (i.e., a path with no repeated vertices) in such a graph is actually an NP-hard problem. Some versions of Dijkstra's algorithm will terminate quickly for such graphs but sometimes they will give incorrect results. This is not the case for our reference implementation. Our reference implementation is actually solving a slightly different problem: for each vertex v, we are looking for the length of the shortest *walk* from the starting vertex to v. (A walk is a path that may contain repeated vertices and edges. Note that if all edge lengths are positive, the shortest walk has to be a simple path.)

Sometimes there is no shortest walk from the starting vertex to some vertex v, because for every walk we can find an even shorter one. For such inputs our reference implementation never terminates.

Problem specification

In this problem, the word "graph" always denotes a directed weighted graph with no duplicate edges and no self-loops. The letters n and m denote its numbers of vertices and edges. Vertices are numbered 0 through $n-1$. The starting vertex is 0.

We want you to show that there are graphs for which Dijkstra's algorithm terminates, but its time complexity is exponential in the number of vertices.

The reference implementation contains a variable named `PROCESSED_VERTICES`. In the **easy subproblem D1**, construct any valid graph (defined below) such that our implementation will terminate on it after finitely many steps, and the value of `PROCESSED_VERTICES` at the end will be at least 10 000.

In the **hard subproblem D2**, you are given several values p. For each p, construct any valid graph for which our implementation will terminate after finitely many steps, and the value of `PROCESSED_VERTICES` at the end will be exactly p.

Input specification

You are given the files `d.cc` and `d.py`. These contain equivalent reference implementations in C++11 and in Python 2.

You are also given the file `d2.in`. The first line of this 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 number p. You may assume that $1 \leq p \leq 10\,000\,000$.

Output specification

In a valid graph, $1 \leq n \leq 60$ and the length of each edge fits into a signed 32-bit integer variable.

Each graph should be output as a sequence of $m+2$ lines. The first of these lines should contain n, the second line should contain m, and each of the following m lines should contain three integers describing an edge – the vertex numbers of its two endpoints (between 0 and $n-1$) followed by its length.

The output for the easy subproblem D1 should contain exactly one such graph. The output for the hard subproblem D2 should contain a sequence of t such graphs. Do not output any empty lines between them.

Example (hard subproblem)

input

```
1

6
```

output

```
12
7
1 3 10
0 7 12
7 11 -4
0 3 9
3 7 1
11 1 12
0 11 9
```

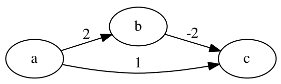
For this graph, our implementation will process exactly 6 vertices: 0, 3, 11, 7, 11 (again), and 1.

Task authors

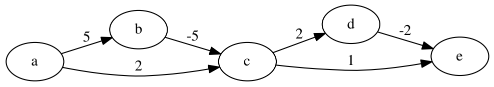
Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Vlado 'usamec' Boža

Solution

Below is the smallest example of a graph that will force Dijkstra's algorithm to evaluate the same vertex (in this case, vertex c) twice. The algorithm will process the vertices in the following order: a at distance 0, c at distance 1, b at distance 2, and then c again at distance 0.



We can now concatenate two such blocks, as follows:



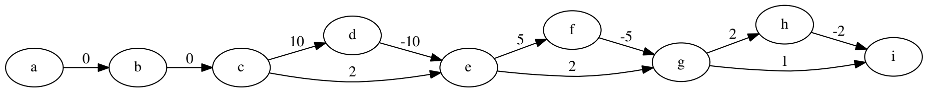
This will force the algorithm to recompute the last vertex 4 times: at distance 3, then 2, then 1, and finally 0. The above pattern can be easily generalized to an arbitrary length; the number of times the last vertex is recomputed will always be 2^t, where t is the number of triangles we used. A long enough pattern like this is sufficient to solve the easy subproblem.

(Note that the edge lengths used in the above construction also grow exponentially in the number of triangles. This is actually necessary in any valid solution to our problem – if all the edge lengths are, say, polynomial in n, all the distances computed during the algorithm will also be polynomial in n, and therefore the total number of updates will also be polynomial in n.)

Tweaking the construction to solve the hard subproblem is not that hard. Here's one possibility: Note the leftmost horizontal edge (edge ac in our picture above). By decreasing its length we can very smoothly decrease the number of vertices processed by the algorithm. In order to construct a graph with a specific number of processed vertices, we can now:

1. Find the smallest number of triangles that produces at least the necessary number of processed vertices.
2. Decrease the length of the first horizontal edge to make the number of processed vertices either exact or slightly smaller than what we need.
3. If necessary, prepend a chain of vertices that will only get processed once, at the beginning.

Below is an example for $p = 20$ processed vertices. Note that the length of the edge ce has been decreased from 4 to 2.

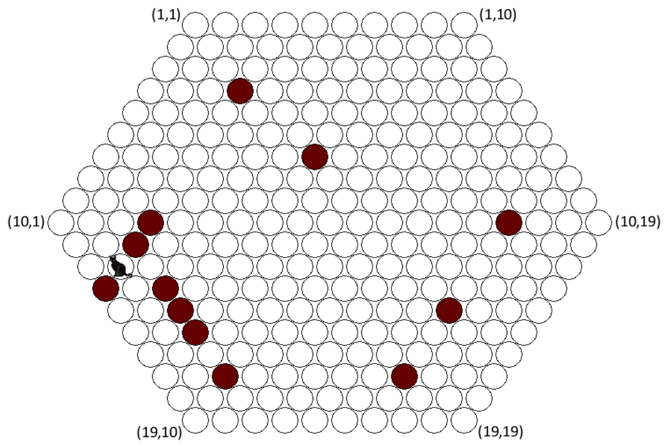


Problem E: Enclosure

"Enclosure" is a simple turn-based game. The game is played by two players. One of them is you, the other is an evil cat. The cat is located on a finite hexagonal grid. Whenever it's the cat's turn, the cat moves from its current hex to one of the neighboring hexes. (The cat cannot remain still, it always has to move.) Whenever it's your turn, you get to click on one of the empty cells to block it off forever. You start the game by blocking the first cell.

The cat wins the game if it reaches the border of the hexagon. You win the game if you block the cat completely – i.e., you win if the cat doesn't have any valid moves left.

A sample situation during the game is shown in the figure below. The blocked cells are filled. The image also shows the coordinate system used in the game. The cat starts in the middle: at $(10, 10)$.



Problem specification

Catch the cat!
(Also see below for the special Cat Challenge!)

Input specification

The cat's moves are completely deterministic. You are given an implementation of the cat's algorithm in the file `catlib.py`. You are also given several tools to help you interact with the cat:

- The script `cat-commandline.py` is a simple command line interface to the library. It alternately reads your move from the standard input and prints the cat's move to the standard output.
- The script `cat-pygame.py` is a simple implementation of the game using the Pygame library.
- Whenever you win or lose a game, the game library automatically appends a log of the game to the file `log.txt`. (So, for example, if you just won a game using the Pygame interface, you can find a log of that game at the end of the log file.)

Output specification

The output for the **easy subproblem E1** should contain any sequence of valid moves that catches the cat. The output for the **hard subproblem E2** should contain any sequence of **at most 17** valid moves that catches the cat. You may use any whitespace.

For a game in which you made m moves, the output should contain a sequence of $2m$ integers: the coordinates of blocked cells, in order. It should **not** contain any parentheses, commas, or the strings "`NEW GAME`" and "`WINNER:`" that are added to the game log by the library.

Cat Challenge!

This problem comes with an additional special challenge! If you are bored with the rest of the problem set, you can keep looking for the shortest solution you can find.

Note that the contest system will only allow you to submit a correct solution to each subproblem once. Once you solve both subproblems, you will not be able to make additional submissions. If you later improve your solution, you may e-mail the new, shorter game log to `ipscreg@ksp.sk` with the subject "Cat Challenge".

At the end of IPSC 2015 we will announce the three teams with the shortest solutions (breaking ties by submission time / e-mail reception time). The top three teams will also be enshrined for all eternity in the solution booklet.

Task authors

Problemsetter:	Michaela 'Šandyna' Šandalová inspired by a flash game 'Chat noir' which is in turn inspired by the Quadraphage game of 'Winning Ways' fame
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Jano Hozza (who is also the author of our best 12-move solution)

Solution

This is intentionally a very open-ended task, with a whole spectrum of possible approaches.

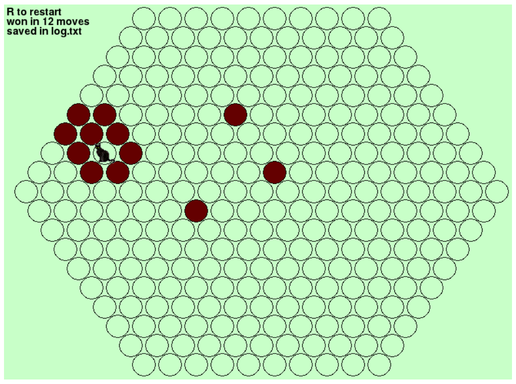
It is possible to view this as an implementation problem. Clearly, all your last moves will be in the proximity of the cat, which makes it reasonably easy to solve the endgame optimally, but what to do with the first part of the game? Maybe making the first few moves randomly, or in some fixed pattern will do the trick.

It is also possible to view this as a code analysis problem. Read the cat's implementation, try to understand its strategy and use that to build a trap. It is our opinion that it never hurts to take at least a cursory glance into the source code. A high-level scan through the library should have told you that the cat follows one strategy for the first 7 turns, and then a different one for the rest of the game. The first strategy is something about free parts of the board, the second strategy is something with shortest paths. Since turn 8, the cat essentially follows some shortest path to the boundary. This knowledge can be quite useful when playing the game.

Which brings us to option 3: treat this task as an actual game. Play it, learn the cat's behavior, adapt, and eventually solve the task completely by hand.

Each of these options has its benefits. You should have chosen the one that plays to your strengths. And which one of these do we prefer? Actually, neither. Our favorite solution involves both writing code and playing the game – but it does *not* involve writing any code that *solves* the game. We did a very different thing. In a few additional lines of code we significantly improved the Pygame implementation. In particular, the two features we added were Save/Load and Undo. (Both can easily be implemented without messing with the library. For example, to undo, just take the log of the current game, start a new game, and replay the old log except for its last item.) Having these, playing the game becomes much more pleasant. You will learn the cat's movement patterns faster, you will be able to try different possibilities for the next move, and take back a bad move without restarting the entire game.

The best solution we were able to find by playing the game in this improved visualizer takes 12 moves: 11 8, 9 10, 6 7, 7 3, 6 2, 6 1, 9 3, 8 2, 8 4, 9 4, 7 1, 7 2. The final configuration is shown below.



Cat Challenge results

Two teams managed to match our 12-move solution. The faster of those two teams and the amazing winner of the Cat Challenge is... wait for it... **Stjupit Dox**: Peter Trebatický, Peter Lacko, and Matúš Horváth. Congratulations!

The second place goes to **NIN Team** who also submitted a 12-move solution, approximately two and a half hours later than our winners.

Six other teams found a 14-move solution. The fastest of those was the team **Open Ports**.

Problem F: Familiar Couples

The rural village Spišský Štvrtok is populated by n married couples – n men and n women, both labeled from 1 to n in such a way that for each i, man i is married to woman i.

Whenever two men meet in the village pub, they become friends. Friendship lasts forever. We say two men are *acquaintances* if they are friends, or if they have a mutual acquaintance. This means that if man a and man b become friends, they also become acquaintances, and all acquaintances of a become acquainted with all acquaintances of b. Similarly, two women can become friends. We say two women are acquaintances if they are friends, or if they have a mutual acquaintance.

Couples living in the village can be familiar with each other. We say couple a is familiar with couple b (they're a "familiar pair of couples") if man a and man b are acquaintances, and woman a and woman b are also acquaintances.

Problem statement

At the beginning, no two people are friends with each other. Then q events happen in sequence. Each event is either a meeting between two men or a meeting between two women. If the two people who met aren't friends yet, they become friends.

After each event, compute the number of familiar pairs of couples. That is, count the number of pairs a, b ($a \neq b$) such that man a is acquainted with man b and woman a is acquainted with woman b. Note that the pairs are *unordered* – for example, $1, 2$ is the same pair as $2, 1$.

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 starts with a line containing the integers n and q. The i-th of the following q lines contains integers t_i, a_i, b_i ($1 \leq t_i \leq 2; 1 \leq a_i, b_i \leq n; a_i \neq b_i$) describing event i. If t_i is 1, man a_i and man b_i meet. If t_i is 2, woman a_i and woman b_i meet.

In the **easy subproblem F1**, $2 \leq n \leq 1\,500$ and $1 \leq q \leq 2\,000$.

In the **hard subproblem F2**, $2 \leq n \leq 1\,000\,000$ and $1 \leq q \leq 1\,000\,000$. Because the input file size for subproblem F2 is about 50 MB, you cannot download it directly. Instead, we have provided a small Python 2 program that will generate the file `f2.in` when executed.

Output specification

For each test case: Let y_i be the number of familiar pairs of couples after event i. Then, let z_i be $i \cdot y_i$. Output one line with a single number: the sum of $z_1 + \ldots + z_q$ modulo $10^9 + 7$.

Example

input

```
1

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

output

```
22
```

After event 3, couple 1 is familiar with couple 3. After event 5, all couples are familiar with each other. Together, we get $1 \cdot 0 + 2 \cdot 0 + 3 \cdot 1 + 4 \cdot 1 + 5 \cdot 3 = 22$.

Task authors

Problemsetter:	Tomáš 'Tomi' Belan
Task preparation:	Tomáš 'Tomi' Belan
Quality assurance:	Michal 'Mimino' Danilák

Solution

When two men become friends, their sets of acquaintances merge into one set. This might remind you of the disjoint-set data structure and the union-find algorithm which can merge two disjoint sets in $O(\alpha(n))$ amortized time. But this is a red herring. In fact, a more "naive" approach works much better in this problem.

Consider the standard problem solved by the union-find algorithm: we have n elements, each initially belonging to a singleton set, and we want to be able to *merge* two sets together and to *find* which set a particular element belongs to. The $O(\alpha(n))$ algorithm uses union-by-rank and path compression, but there is a simpler solution.

For each element, we will remember the ID of the set it belongs to, and for each set we will remember the list of its elements. Thus, *find* is $O(1)$. When we want to *merge* two sets, we simply take the smaller of the two, iterate over its elements, and move them to the other set one by one. This trivial algorithm actually achieves $O(n \log n)$ total merging time. This is because we always move elements from a smaller set to a bigger set, so when an element moves, its new set is always at least twice as big as the old one. Thus, each element can move at most $\log n$ times.

Let's use this idea in our problem. Each couple will have a pair (m_i, w_i), where m_i is the ID of the set the man belongs to, and w_i is the ID of the set the woman belongs to. So two men i and j are acquaintances if $m_i = m_j$, and two couples i and j are familiar if $(m_i, w_i) = (m_j, w_j)$.

When a man belonging in set a and a man belonging in set b become friends, we have to merge the two sets. First, we find which of them is bigger – for example the set containing a. Then we just iterate over each couple where $m_i = b$, and change it to a. For women, we do the same. Like above, we still move each couple at most $O(\log n)$ times.

We can do this easily if, for each a, we keep a list of all couples where $m_i = a$, and a list of all couples where $w_i = a$. We update these lists whenever we move any couple from one set to another. Similarly, for each a and b, we remember the number of couples with these particular numbers (a, b). Whenever this number increases, we change the total number of familiar pairs of couples appropriately.

Problem G: Generating Synergy

You are working for IPSCorp, a multinational corporation which is revolutionizing the world by providing end-to-end solutions for high-impact paradigm shifts. This important task requires that key players touch base with industry leaders to incentivize core competencies and faciliate organic growth.

As you can probably guess from this description, most of IPSCorp consists of layers and layers of incompetent managers. Since they need to look productive, they spend their workdays attending endless meetings and sending pointless memos to their subordinates. Making business decisions is hard, so the memos are always about trivial issues such as the office dress code.

When a manager writes a memo, he only sends it to his direct reports (i.e. subordinates of which he is the direct supervisor). They read the memo and forward it to their own direct reports. This way, the memo goes deeper and deeper in the company hierarchy. But after the memo has been forwarded a certain number of times, the recipients will just ignore it, hoping that the original author won't notice. This depends on the memo's general tone, number of exclamation marks, firing threats, and so on. We call the maximum number of forwards the "importance level".

A memo of importance level 0 ("I think we should wear red ties.") only affects the memo's author – he starts wearing a red tie, but even his direct reports ignore it. A memo of importance level 1 ("I want everyone in this room to wear blue ties from now on.") affects the author and his direct reports, but nobody else cares. A memo of importance level 2 ("New department policy: only green ties allowed!") also affects the direct reports of the author's direct reports. And so on.

Problem description

IPSCorp consists of n employees, labeled from 1 to n and organized in a hierarchical tree. All employees own ties of c colors labeled from 1 to c.

At the beginning, everyone is wearing a tie of color 1. Then, q events happen. Events are of two types: Either a given person writes a memo of a given importance level, and its recipients (including the person who wrote it) start wearing ties of a given color, or we want to know what tie a given employee is wearing at the moment.

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.

Every test case starts with a line containing the integers n, c and q. The next line contains $n-1$ integers named s_2 to s_n, where s_i is the supervisor of employee i ($1 \le s_i < i$). Employee 1 is the company president and has no supervisor.

The i-th of the following q lines contains three integers a_i, l_i, c_i describing event i ($1 \le a_i \le n; 0 \le l_i \le n; 0 \le c_i \le c$). If c_i is nonzero, it means person a_i sent a memo of importance level l_i saying the new tie color is c_i. If c_i is zero, then l_i is also zero, and it means you have to find the tie color worn by person a_i.

In the **easy subproblem G1**, $1 \le n, c, q \le 10\,000$.

In the **hard subproblem G2**, $1 \le n, c, q \le 1\,000\,000$. Because the input file size for subproblem G2 is about 100 MB, you cannot download it directly. Instead, we have provided a small Python 2 program that will generate the file `g2.in` when executed.

Output specification

For each test case: Let y_i be equal to 0 if c_i is nonzero, or the tie color you found in event i if c_i is zero. Then, let z_i be $i \cdot y_i$. Output one line with a single number: the sum of $z_1 + \ldots + z_q$ modulo $10^9 + 7$.

Example

input

```
1

4 3 7
1 2 2
3 0 0
2 1 3
3 0 0
1 0 2
2 0 0
4 1 1
4 0 0
```

output

```
32
```

At the beginning, everyone has tie color 1 (including employee 3). Then, employees 2, 3 and 4 change to color 3. Employee 1 then changes to color 2, but because his memo had importance level 0, everyone else stays unchanged. Finally, employee 4 changes his color back to 1. His direct reports would change too, but he doesn't have any. Together, we get $1 \cdot 1 + 2 \cdot 0 + 3 \cdot 3 + 4 \cdot 0 + 5 \cdot 3 + 6 \cdot 0 + 7 \cdot 1 = 32$.

Task authors

Problemsetter:	Tomáš 'Tomi' Belan
Task preparation:	Tomáš 'Tomi' Belan
Quality assurance:	Jano Hozza

Solution

First, consider the case where the input tree T is just a linear list – i.e. each node has at most one child. We can solve this case by building a segment tree S. Each leaf of S corresponds to one node of T, and the internal nodes of S correspond to longer intervals of nodes in T – from some node a to some node b.

Using this segment tree, we can quickly answer the given queries. When we need to update an interval $[a, a+l]$ of nodes in T, we map this interval to $O(\log n)$ nodes of S which exactly cover the original interval in T. We store the "last modification time" (the query number) in these $O(\log n)$ nodes.

When we want to find the color of a particular node in T, we look at the $O(\log n)$ nodes in S which cover it and find the one with the most recent "last modification time". This tells us which query was the last to update this node.

Let us extend this solution to arbitrary trees T. Now there can be multiple nodes on the same level (i.e. with the same distance to the root). Again, we will build a segment tree S. But this time, each leaf of S corresponds to a *level* in T, and each internal node corresponds to an *interval of levels* in T – from level a to level b.

In the linear list case, each leaf of S stored a single number. But now that each leaf of S corresponds to a whole level in T, which might contain more than one node, that won't be enough. Instead, each leaf v of the segment tree S will contain a smaller nested segment tree S_v. The leaves of S_v will correspond to the individual nodes on this level of T. We can use this nested segment tree to quickly update multiple nodes of T which are on the same level.

What if we want to update multiple levels of T together? That's where internal nodes of S come in. Consider an internal node v which manages levels a to b. When you look at only these levels of T and ignore all other nodes, the tree T falls apart into a number of subtrees[1] – as many as there are nodes in level a. These subtrees of T are what will be managed by the nested segment tree S_v (which is stored in node v of segment tree S). So each leaf of S_v corresponds to one subtree, and each internal node of S_v corresponds to an interval of multiple subtrees.

This allows us to efficiently process queries on the tree T. When we need to update l levels of the subtree under a given vertex a, we split the affected area vertically into $O(\log n)$ segments that correspond to nodes of S, and in each segment, we just update some of the subtrees managed by S_v. Analogously, when we want to find the color of a node in T, we find all the nodes v in S which are relevant to it, and then we look in each of the relevant nodes of their nested segment trees, returning the color of the most recent query. This means every query can be completed in $O(\log^2 n)$ time.

[1]Technically, these are not subtrees, because we cut off their descendants below level b. They're vertex-induced subgraphs of the tree T. We call them subtrees for clarity's sake.

Problem H: Humble Captains

Every day just after school n children rush out of their classrooms onto the field to play football. They choose two captains among themselves who then divide the remaining children into two teams. The teams do not need to be of the same size – in the extreme case, an overconfident captain may challenge all the other children to join their forces against him! Adam and Betka are the captains for today's game.

Problem specification

There are m pairs of children who are friends. Two friends playing for the same team are more likely to pass the ball to each other than two non-friends, so they increase the strength of their team. The total strength of a team can therefore be defined as the number of pairs of friends within that team.

Adam thinks a football match is fun when the players score many goals. He would like to **maximize the sum** of the two teams' strengths. On the other hand, Betka believes the most enjoyable matches are balanced ones, so she wants to **minimize the difference** between the teams' strengths.

Find the largest possible sum of teams' strengths and the smallest possible absolute difference between teams' strengths. (These are two independent problems – there may not necessarily be a team division that satisfies both criteria.)

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.

The first line of each test case consists of integers n ($n \geq 2$) and m ($0 \leq m \leq n \cdot (n-1)/2$). Children are labelled $1, \dots, n$. Adam has number 1 and Betka number 2. Each of the following m lines contains two integers u_i, v_i ($1 \leq u_i < v_i \leq n$) meaning that u_i and v_i are friends. Each pair of friends is only listed once.

In the **easy subproblem H1** we have $n \leq 24$.

In the **hard subproblem H2** we have $n \leq 200$.

Output specification

For each test case, output a single line with two integers: the largest possible sum of teams' strengths, and the smallest possible difference between the teams' strengths.

Example

input

```
2

3 3
1 2
2 3
1 3

3 1
1 3
```

output

```
1 1
1 0
```

In the first example, it does not matter whether Cyril (child 3) will play with Adam or with Betka. Either way, one of the teams will have strength 1 and the other 0, so the sum of strengths is 1 and their difference is 1. In the second example, letting Cyril play with Adam increases the sum of strengths, but letting him play with Betka keeps the teams more balanced.

Task authors

Problemsetter:	Peter ‘Bob’ Fulla
Task preparation:	Peter ‘Bob’ Fulla
Quality assurance:	Vlado ‘usamec’ Boža

Solution

For the easy subproblem a simple brute-force solution was sufficiently fast. Let us denote Adam’s team by A and Betka’s team by B. Each of the other $n-2$ children joins either A or B, so there are 2^{n-2} ways to form the teams. We can afford to try them all, every time computing the strengths of the teams in $O(m)$ by processing the list of friendships.

To solve the hard subproblem we will consider Adam’s and Betka’s teams composition preferences separately. Let us start with Adam – he wants to maximize the sum of the teams’ strengths. If we regard pairs of friends as edges of a graph, his criterion corresponds to finding a partition of vertices into two sets A, B which *maximizes* the number of edges *within* each part. This is of course equivalent to *minimizing* the number of edges *between* A and B. Our task is therefore to find a minimum s-t cut where the source and sink are the vertices corresponding to Adam and Betka respectively. There are more efficient algorithms to solve this problem, but even our lazy implementation of the Ford-Fulkerson algorithm with time complexity $O(nm)$ was fast enough.

Now we move on to Betka’s criterion – she would like the absolute difference between the teams’ strengths to be as small as possible. Let us consider any partition of the vertices into teams A and B. For any edge and any of its two endpoints, we add to a running total $1/2$ if the endpoint is in A and $-1/2$ if it is in B. Clearly, the final value we obtain equals

$$\frac{1}{2} \cdot \left(\sum_{v \in A} d_v - \sum_{v \in B} d_v \right)$$

where d_v denotes the degree of vertex v. Moreover, this value is exactly the strength of team A minus the strength of team B: For any edge within A we have a 1, for any edge within B we have a -1, and for any edge between A and B we have a 0.

This observation offers us another way how to phrase the problem. We can represent each vertex by a single number (its degree) and then look for a partition which minimizes the absolute difference between the sums of the numbers in each part. Equivalently, we want to find a subset of the vertices such that its sum is as close to half the total sum as possible. We could compute which sums it is possible to achieve by listing all subsets, but using dynamic programming is much faster: If we denote by $S = \{s_1, \ldots, s_{|S|}\}$ the set of sums obtainable from numbers $d_1, \ldots, d_{k-1}$, then the sums obtainable from $d_1, \ldots, d_{k-1}, d_k$ are $S \cup \{s_1 + d_k, \ldots, s_{|S|} + d_k\}$. Because the sum of all degrees equals $2m$, each set has size $O(m)$ and our solution runs in time $O(nm)$.

Problem I: Inexpensive Travel

Cycling is a cheap and healthy way of getting from A to B, plus it is a lot of fun. That is why Peter invited Kamila to join him on an ambitious bike trip around the world. Unfortunately, cycling can sometimes be hard – just imagine having to ascend 1500 meters on some winding road to a mountain pass in the Alps. Different cyclists have a different degree of preference for the ascents. On one side of the spectrum there is Kamila who would like to avoid going uphill completely. On the other side you will find Peter who thinks that flat roads are painfully boring. Now Peter and Kamila have a problem – they need to figure out which route to choose so that they would both enjoy the trip. Help them by finding all the options they have.

Problem specification

There are n towns in the world. The towns are numbered 1 through n. Peter and Kamila want to start the trip in town 1 and end it in town n. Each road is a directed edge from one town to another. Each road has two parameters: the distance $d \geq 0$ between its endpoints, and the ascent $a \geq 0$. At least one of them is nonzero.

Each cyclist can be described by a single real number $p \in [0, 1]$: their preference for ascents. The value p determines the actual length of each edge for this particular cyclist: an edge with distance d and ascent a will have the length $pd + (1-p)a$. For example, Kamila has $p = 0$ and only cares about ascents. Peter has $p = 1$: he ignores all ascents and only cares about the total distance.

Different values of p will obviously produce different shortest paths from 1 to n. Your task is to find all values p where the set of shortest paths changes.

Formally, let $S(p)$ be the set of all paths from 1 to n that have the shortest total length for a cyclist with preference p. Find all values p such that $0 < p < 1$ and the sets $S(p-\varepsilon)$ and $S(p+\varepsilon)$ differ for any $\varepsilon > 0$. (The two sets differ if there is some specific path from 1 to n that is among the shortest paths in one setting but not in the other one. Note that $p = 0$ and $p = 1$ are by definition never valid answers.)

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 starts with a line containing two integers n and m: the number of towns and the number of roads. Next, m lines follow. The i-th of these lines will contain four integers: x_i, y_i, d_i, and a_i ($1 \leq x_i \leq n$; $1 \leq y_i \leq n$; $0 \leq d_i$; $0 \leq a_i$; $0 < d_i + a_i$). These represent a directed road from the town x_i to the town y_i, with distance d_i and ascent a_i. Note that some of the roads may be self-loops (with $x_i = y_i$) and that each pair of towns can be connected by multiple roads in each direction.

In the **easy subproblem I1** you may assume that $n \leq 100$, $m \leq 5000$, and $0 \leq d_i, a_i \leq 10\,000$.

In the **hard subproblem I2** you may assume that $n \leq 5000$, $m \leq 500\,000$, and $0 \leq d_i, a_i \leq 500\,000$.

Because the input file size for subproblem I2 is about 100 MB, you cannot download it directly. Instead, we have provided a small Python 2 program that will generate the file `i2.in` when executed.

Note that the implementation of this generator might matter.

Output specification

For each test case, find the number v of valid answers and their values $p_1 < p_2 < \cdots < p_v$. Output a single line containing the integer v followed by the real numbers p_1 through p_v, in order. Output each p_i to at least 10 decimal places. Any answer that differs from the correct solution by at most 10^{-9} will be accepted.

Note that in some test cases the set of shortest paths never changes. In such case we have $v = 0$ and thus the output line will contain just a single zero. Notably, this includes all test cases in which it is impossible to get from 1 to n. (In those test cases, each set $S(p)$ is empty.)

Example

input

```
3

2 2
1 2 1 1
1 2 3 0

2 2
1 2 1 0
1 2 1 0

2 0
```

output

```
1 0.333333333333
0
0
```

In the first test case there are two roads from 1 to 2. The first road (distance 1, ascent 1) goes across a small hill, the other (distance 3, ascent 0) takes a detour around the hill. The preference between these roads changes at $p = 1/3$: any cyclist with $p > 1/3$ will prefer the first road while any cyclist with $p < 1/3$ will prefer the second one.

In the second test case there are two different roads, but they look the same to any cyclist. Each of them is a shortest path for any value of p.

Finally, in the third case it is impossible to reach the destination.

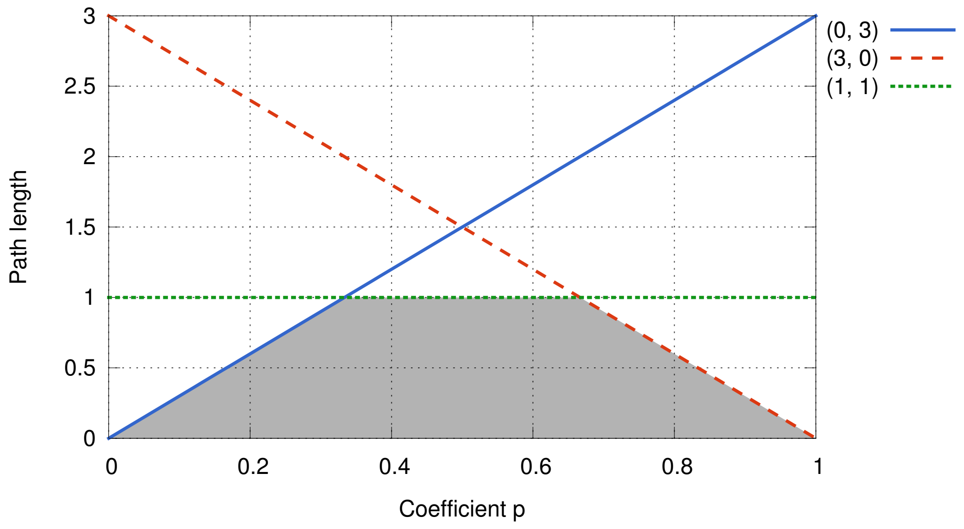
Task authors

Problemsetter:	Peter 'PPershing' Perešíni
Task preparation:	Peter 'PPershing' Perešíni
Quality assurance:	Vlado 'usamec' Boža

Solution

Easy subproblem

Let us start by examining how the path length depend on the parameter p. For example, consider three parallel roads with $(d, a) = (0, 3)$, $(3, 0)$, and $(1, 1)$, respectively. The plot of the shortest length as a function of p is shown in the following figure:



In essence, for each of the roads the length is a linear function of p, hence its plot is a line. The shortest length then corresponds to the lower envelope of these lines – i.e., to the boundary of the intersection of the lower halfplanes determined by our lines. The points p_i where a change occurs are the vertices of this polyline.

This would be enough to solve the problem for very small graphs. You could simply enumerate all simple paths from start to finish, for each of them you could compute the total distance and the total ascent, and then you would have as many lines as there were simple paths. The vertices of the lower envelope for those lines would be the answer we seek.

Now we have to make an observation that is similar to how the shortest paths algorithms operate: whenever a path from x to y is "too long", i.e., if it is not the shortest path for at least one value $p \in (0, 1)$, adding an edge $y \to z$ will always create a path from x to z that is "too long".

Therefore, as a general recipe, we can discard "too long" paths as soon as we can determine that we don't need them. Now, the question is how to construct all the remaining paths. Here, we could again re-use ideas from the standard shortest-path algorithms: These algorithms basically iterate over edges and for each edge $x \to y$ they try to shorten the best solution for y using the best solution for x. We can do the same, just instead of a single number (length) we need to remember all combinations of (d, a) that are not "too long". However, there is a potential caveat here – algorithms such as Dijkstra rely heavily on the particular order in which vertices are processed, and such order is hardly going to be consistent for different distance-vs-ascent tradeoffs. Fortunately, there is a shortest paths algorithm that is oblivious to the ordering – Bellman-Ford. This algorithm just iterates n times over all edges in an arbitrary order. The whole solution is therefore to run Bellman-Ford over all edges, for each edge $x \to y$ compute the union of all paths to y with all paths to x extended with the edge $x \to y$, and then throw away all "too long" paths from this set by computing the geometric intersection. The intersection is a standard geometric algorithm that has an efficient implementation using a sweep line.

Hard subproblem

If we look at the hard subproblem, it looks pretty hard – the limits are really big and there does not seem to be an easy way to improve our previous solution to handle them.

However, it wouldn't be a real IPSC contest if we did not prepare something special for you. IPSC is an open-data contest – you have the data, examine it. (We even suggested that in the problem statement. Or did you skip that sentence?)

So, what can we learn from the input generator? Let's look closely at the following code:

```
x = rng.randint(0, n-2)
y = rng.randint(x+1, n-1)
...
print permutation[x], permutation[y], dist, hill
```

Clearly, x and y are generated in such a way that $x < y$. Therefore, the graph generated by this algorithm is a directed acyclic graph. Computing shortest paths is much easier on such graphs – we just iterate (once) over all vertices in the topological order and we are done.

Oh wait, but now there is this ugly permutation. Does that mean I have to implement topological sort as well? No, you don't have to. The really cool thing about input generators is that we may actually modify them. In other words, we can print the permutation to the output as well. That way, we do not need to recompute it, just use the already pre-computed value.

Not simple enough? Just get rid of it completely: change the print line to

```
print x+1, y+1, dist, hill
```

and the permutation is gone. (Note that you still have to *generate* the permutation because you do need to consume the output of the random generator.)

Problem J: Juicy Dot Coms

As you shall soon discover, the dot coms from the title of this task have nothing to do with the dot com boom. These are some dot coms from the previous generation.

You are given the files `j1.com` and `j2.com`. They know the passwords. The easy one might just tell you the password, but it may take some convincing. The hard one will be, of course, harder. It seems… who knows, broken? Some repairs may be in order.

Oh, and there's an obvious trap in the easy one. Don't fall into the trap. You'll know the right password once you get it.

Problem specification

The password is a sequence of upper- and lowercase English letters. Recover the password.

Input specification

The input is the corresponding dot com file.

Output specification

For each test case, your output should contain a single line with the password.

Task authors

Problemsetter:	Peter ‘goober’ Košinár, Michal ‘mišof’ Forišek
Task preparation:	Peter ‘goober’ Košinár
Quality assurance:	Michal ‘mišof’ Forišek

Solution

When dealing with an unknown file format, the extension of the filename can be a good hint. Wikipedia is quite helpful when it comes to .COM files.

Once you found out that it’s an old MS-DOS binary, you probably wanted to run it – but didn’t have a copy of MS-DOS lying around any more. Luckily for you, DOSBox runs almost everywhere, and can run these binaries as well. That was actually sufficient to solve the easy subproblem, as described below.

Hollywood hacking

The easy subproblem was actually solvable without any need for understanding how computers work – almost like hackers break passwords in movies. The program gives immediate feedback when an incorrect character is pressed. This means we can figure out the password on character-by-character basis, simply by entering the known prefix of the password and exhaustively trying to extend it by one more character. Once you discovered that the program *doesn’t* terminate if you press an uppercase A, you probably knew you were on the right track.

Still, it was beneficial to look at the “code” as well, as the easy subproblem could prepare you for the hard one.

16-bit machine code basics

Apparently, the files consist of raw machine code for a 16-bit processor. There are many ways of turning the machine code into something more readable; a few can be found in this Stack Overflow answer.

In order to understand the dissassembled output, one would need a quick course in x86 assembler. It’s Wikipedia to the rescue again: X86 assembly language!

For our purposes, it would be sufficient to understand that we are dealing with a few 16-bit values (so-called “registers”) which are named `ax`, `bx`, `cx`, `dx`, `si`, and `di`. Each of them can be split into two 8-bit halves: e.g., the higher 8-bit half of `ax` is called `ah`, and the lower 8-bit half is called `al`.

Most of the instructions are named reasonably enough to infer their meaning from the name. The syntax `[X]` denotes access to memory location `[X]`; similar to dereferencing a C-style pointer.

The only tricky instruction is “`int 0x21`”, which was, back in the MS-DOS days, used for calling various functions provided by the operating system – such as reading input from the user or writing output. The definitive guide to these functions (and a lot more) is Ralf Brown’s Interrupt List (RBIL).

Disassembling the easy subproblem

Now, we are ready to do some disassembling! From one of the above links, we have learned that the .COM file is loaded into memory so that its first instruction is at the address 100h. Hence, we may want to run a command like this one:

```
$ ndisasm -b16 -o100h j1.com
```

The output starts with:

```
00000100  BA3001            mov dx,0x130
00000103  B409              mov ah,0x9
00000105  CD21              int 0x21
```

RBIL tells us this block just writes out a piece of text at the address that is stored in the `dx` register. A quick look into the position 0x30 of our file (remember, the first byte of the file is at address 0x100 in memory) makes it clear it is just the introduction message. Moving on.

```
00000107  BE2002            mov si,0x220
0000010A  BF5E02            mov di,0x25e
0000010D  B90A00            mov cx,0xa
00000110  BA0102            mov dx,0x201
```

Similar reasoning allows us to find that `si` now points to the string "`7ialnjpbjd`", `di` to "`Pythagoras`" and `dx` points to the message displayed after entering an incorrect password. Finally, `cx` has the value 10, which is suspiciously similar both to the length of the strange string and to the philosopher's name.

If we now skip ahead a bit, we'll see the following sequence of instructions:

```
0000011F  46                inc si
00000120  47                inc di
00000121  49                dec cx
00000122  75EF              jnz 0x113
```

The last two instructions are essentially a do-while loop: the register `cx` is decremented by one and if it didn't hit zero yet, execution continues back at the address 0x113. Thus, `cx` serves as a loop counter, starting at 10 and decreasing with every iteration. At the same time, the registers `si` and `di` move forward over the respective strings they pointed at initially.

The instruction immediately following the loop is

```
00000124  BA3A02            mov dx,0x23a
```

and a quick check of the corresponding location in the file tells us we're on the right track – the position 0x13a corresponds to the success message we want to see! The only way of reaching this point in the program is to run the loop through all the 10 iterations.

The body of the loop looks as follows:

```
00000113  B401              mov ah,0x1
00000115  CD21              int 0x21
00000117  8805              mov [di],al
00000119  28C8              sub al,cl
0000011B  3A04              cmp al,[si]
0000011D  7508              jnz 0x127
```

RBIL tells us that first two instructions just cause one character to be read from input. This character is then used to overwrite one character of "Pythagoras" and it is compared to one character of that strange string – but only after the current value of the loop-interator is subtracted from it.

If they differ (their difference being "non-zero", thus `jnz` = "jump if not zero"), the program continues at location 0x127. Quick glance at that location reveals that it just contains one display-a-message request followed by program termination:

```
00000127  B409              mov ah,0x9
00000129  CD21              int 0x21
0000012B  B8004C            mov ax,0x4c00
0000012E  CD21              int 0x21
```

Since we need to go through all the iterations of the loop, we need to make sure the comparison always ends up true. This means the first character we input must be equal to the first character of the "`7ialnjpbjd`" string plus 10, the second one must be equal to the second character of this string plus 9, and so on. Adjusting the characters in the string reveals the password to be "`Aristotele`".

(And indeed, if you enter `Aristotele` from your keyboard into the program, it will verify it and output a confirmation that this is indeed the password.)

Decompiled easy

Here's some pseudocode that is roughly equivalent to `j1.com`:

```
    write("... welcome...");

    char *pkey = "7ialnjpbjd"; // register "si"
    char *presult = "Pythagoras"; // register "di"
    int count = 10; // register "cx"
    char *message = "... failed..."; // register "dx"

    do
    {
        char al = read_character_from_input();
        *presult = al;
        if (al - count != pkey)
            goto finish;
        pkey++;
        presult++;
        count--;
    } while(count > 0);
    message = "... success..."

finish:
    write(message);
    exit(0);
```

Disassembling the hard subproblem

The hard sub-problem is slightly more complicated, but if you didn't follow the movie-hacker path, you should be ready for it! Unlike the easy case, this program doesn't take any input; it just displays three strings and terminates. The second of those messages is displayed in a subroutine at address 0x288 and mentions the program as being "cracked". A subroutine is entered using "call" instruction and finishes when the execution reaches the "ret" one. Looking carefully, we can see that there is a large block within the program which looks reasonable enough, yet is never entered. This block starts at address 0x116 and seems to try to write some strings – which turn out to be mentioning "license". Last but not least, it contains a "ret" (at 0x15c). All of these observations suggest this block is the license-checking function which the wannabe-cracker replaced by a call to his own code.

As the first step, let's "un-crack" the program by modifying the "call" instruction at address 0x107. Call instruction consists of three bytes; the second and third of which determine the address to be called – they represent a signed 16-bit value which corresponds to a relative offset between the next instruction and the desired address to be called. In the cracked program, these two bytes represent value 0x17e (0x10A + 0x17E = 0x288). Replacing them by 0x10c (i.e. 0x0c, 0x01) changes the call destination to 0x10A + 0x0c = 0x116, just as we desired.

Doing this removes the crack… but of course, if you now run the code, it produces a complaint about license information not being valid.

Let's see how the license is validated. A quick look tells us that the whole validation process takes place in the block starting at address 0x11d and if we reach address 0x139 in a legitimate, non-cracky way, we have won – since that's where a message about license being valid is displayed. Conditional jumps leading to address 0x10d correspond to failed checks – if a program jumps to that address, it will only display a message and terminate.

Let's see what those conditions are!

```
0000011D  A13302           mov ax,[0x231]
00000120  8B1E3502         mov bx,[0x233]
00000124  BA2402           mov dx,0x220
// Okay, we will be dealing with two 16-bit values: ax and bx
// "dx" is only used for the message about license being bad.

00000127  B117             mov cl,0x17
00000129  39C3             cmp bx,ax
0000012B  72E0             jc 0x10d
0000012D  83F800           cmp ax,byte +0x0
00000130  74DB             jz 0x10d
00000132  29C3             sub bx,ax
00000134  93               xchg ax,bx
00000135  FEC9             dec cl
00000137  75F0             jnz 0x129
```

Translating this block into a pseudocode yields (writing A and B instead of `ax` and `bx`):

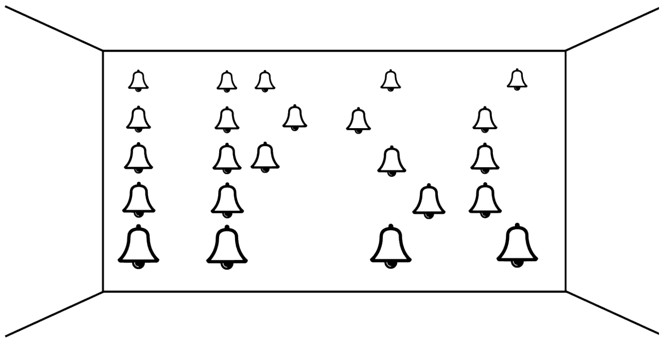
```
for (count = 23; count != 0; count--)
{
    if (B < A || A == 0)
        fail;
    B -= A;
    swap(A, B);
}
```

Since we are dealing with 16-bit values, it is easy to determine the correct values of `A` and `B` by exhaustive search and find that the only pair which survives through the 23 iterations is `A` = 28657 and `B` = 46368. Just out of curiosity… do you happen to know what the 23-rd Fibonacci number is? No, it's most definitely not a coincidence.

Finally, we need to place these numbers at offset 0x131 in the file (that's where `ax` and `bx` were initialized from), overwriting the conveniently located string `IPSC`. We have 0x6ff1 = 28657, while 0xb520 = 46368, so the final byte-level representation would be 0xf1, 0x6f, 0x20, 0xb5 (remember, we are working with a little-endian platform). Doing so yields the final executable which, upon execution, reveals the answer: "`LeonardoPisano`".

Problem K: Klingelt das Glockenspiel

You have just bought an amazing new Carillon: a musical instrument that consists of bells of various sizes. You arranged all the bells into an interesting pattern and you mounted them onto your living room wall. While doing so, you followed two simple rules. First, the bells could only be mounted at regularly spaced grid points. Second, the smaller the bell, the higher you mounted it. For example, it could have looked like this:



You then sat into your armchair facing the wall with all the bells. You relaxed, closed your eyes, and enjoyed the music.

Problem specification

In each subproblem, you are given a different scenario that is consistent with the above description. You are given a recording of the bells. Listen to the performance and identify a short English word you should submit as your answer.

In the easy subproblem you will quickly discover that the music follows a nice regular pattern. The hard subproblem, on the other hand, is just pure chaos.

Input specification

The input file is a stereo MP3 file of the recording.

Output specification

Output a single line with the English word determined by the recording. The word should be written in UPPERCASE.

Task authors

Problemsetter:	Michal 'mišof' Forišek, inspired by a puzzle idea by Michal 'Kesy' Kesely
Task preparation:	Vlado 'usamec' Boža
Quality assurance:	Michal 'mišof' Forišek

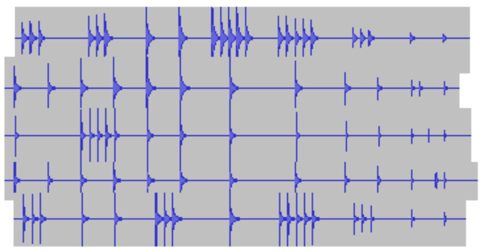
Solution

Easy subproblem

In the easy subproblem we went through the grid in row major order. Whenever we encountered a bell, we rang it. Just by listening you could determine that there are five rows of bells. If you then just took silent and non-silent segments as zeros and ones, you could easily determine the locations of bells. The grid of bells looks as follows:

```
 ###     ###    #   #   #####   #####    ###    #   #
#   #   #   #   #   #     #       #     #   #   ##  #
#       #####   #   #     #       #     #   #   # # #
#   #   #   #   #   #     #       #     #   #   #  ##
 ###    #   #    ###      #     #####    ###    #   #
```

There were many ways how to actually determine the location of bells. For instance, instead of listening to the recording and decoding them manually, you could open the sound file in a sound editor, take a screenshot of the waveform, and then arrange the parts for each bell size below one another. The figure below shows the waveform for the left ear arranged this way. Note that in each row the sounds are getting fainter as we are getting close to the right end of the wall.



Hard subproblem

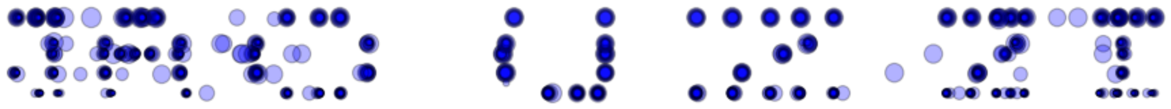
In this subproblem, each bell rings once and bells ring in random order. So we have to determine the pitch and position of each sound. This is not that hard. The pitch can be found using the Fourier transform and the position can be found by finding the amplitude differences between left and right channel. (Note that you can use the easy input to test your implementation, and as an information source that tells you how the amplitudes change while moving from left to right.)

In our solution we:

1. Split the file into segments of length 0.1 s. Drop segments which are too loud.
2. For each segment determine the mean absolute values of left and right channel. Denote them L and R. We calculate position as $P = (L - R)/((L + R)/2)$.

3. Do the Fourier transform of each segment. Let F be the first frequency which is loud enough.
4. Visualize each segment as a bubble where the x-position is $P * abs(P)$ (for better visualization) and the y-position is F. We also make the size of bubbles proportional to the intensity of the sound.

After that we get following result:



Even with the noise we can see that the answer is JACUZZI. The third letter is indeed C and not O (bubbles on the left edge have three different y-positions but bubbles on the right edge have two).

Bell sounds

The bell samples were created by angstrom and are available online under a Creative Commons 0 license. Thanks!

Problem L: Lunchtime!

A foreign restaurant recently opened in your area. You like the food they make, so you convinced your friends to start having lunch there each day. But all the dishes have foreign names and you don't know which is which. You can only identify a dish after you order it.

There are p people in your group (including yourself) and the restaurant serves d different dishes. Ordering at the restaurant works as follows: First, each person orders exactly one dish by specifying a number between 1 and d. Then, the p dishes ordered by the group are brought to the table all at once, in no particular order and with no information on which is which.

Your goal is to create a menu in your language: you want to map all numbers between 1 and d to names of dishes in your language. Assuming that your friends are willing to cooperate, what is the smallest number of lunches in which this can be done?

Problem specification

Find the smallest number of lunches ℓ such that there is a strategy for placing the orders in such a way that after ℓ lunches you will know the names of all dishes. In the **hard subproblem L2**, you also have to find the number of valid first day orders.

An order is a multiset of dish numbers your group ordered. For example, if $p = 3$ and $d = 2$, the order $\{1, 1, 2\}$ is the same as the order $\{2, 1, 1\}$ but different from the order $\{1, 2, 2\}$. A valid first day order is an order such that if your group makes the order on the first day, there will be a strategy that solves our task by making $\ell - 1$ additional orders.

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 p and d.

In the **easy subproblem L1** we have $t = 81$, $1 \le p \le 9$, and $1 \le d \le 9$.

In the **hard subproblem L2** we have $t = 320$, $1 \le p \le 16$, and $1 \le d \le 20$.

Output specification

Output a single line for each test case.

In the **easy subproblem L1** the line should contain a single integer: the minimum number of days.

In the **hard subproblem L2** the line should contain two integers: the minimum number of days, and the number of valid first day orders.

Example (hard subproblem)

input

```
3

1 1

2 3

6 3
```

output

```
1 1
2 3
1 6
```

In the first example, note that the answer is not 0: you have to order the dish once to see what it is.

Here's one optimal strategy for $p = 2$ and $d = 3$: On the first day, one of you will order dish 1 and the other will order dish 2. On the second day, order dishes 1 and 3.

In the third example order one dish 1, two dishes 2, and three dishes 3.

Task authors

Problemsetter:	Michal 'mišof' Forišek
	inspired by Evan Morton's IBM Ponder This September 1999 challenge
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Michal 'Mimino' Danilák

Solution

First of all, we should realize that there is no need for an adaptive strategy. In other words, we don't have to see the "outcome" of day 1 before determining the optimal order on day 2. This is because there isn't actually any hidden outcome. We know what we'll get as soon as we make the order. For example, if the first day order is $(1, 1, 2, 3, 4)$, we know that after we get the meals:

- we will know the name of dish 1
- we will know the three names of dishes 2, 3, 4, but we won't know which is which
- we will know nothing about the other dishes (if any)

Thus, the entire optimal strategy can be determined in advance, as a sequence of ℓ fixed orders.

How can we tell whether a sequence of orders solves our problem? First of all, each dish must have been ordered at least once. Second, we need to be able to tell each pair of dishes apart. In other words, for each pair of dishes there has to be a day when we ordered different amounts of the two dishes.

This observation alone turns out to be sufficient to solve the easy subproblem. The constraints are small enough, so if you try a sufficient amount of random orders, you are virtually certain to discover one that's optimal – there are always many of them.

Easy subproblem, deterministic solution

Of course, we can also solve the easy subproblem in a deterministic way: by traversing the entire state space. Imagine any situation where you already spent some days on making orders. How does the information on the dishes look like? The dishes are divided into some *equivalence classes*. Each equivalence class is a set of dishes we still cannot tell apart – because so far we ordered the same quantity of each of them on each day. For example, after three days there will be one (possibly empty) equivalence class of dishes such that you ordered once on day 1, not at all on day 2, and three times on day 3.

Each division of dishes into equivalence classes represents one possible state. And for each state we want to answer the same question: "What is the smallest number of additional days we need to get from this state to the end of the game?"

This can be done using recursion with memoization. To solve a specific state that is not the final state, try all possible orders for the next day. Each of them will bring you into some new state. Solve each of those recursively, and then your answer is 1 + the minimum of their solutions.

There are several things to note here. First, when processing a given state we do not actually need to know the sequence of orders that led to the state. All the information is already contained in the division of dishes into equivalence classes.

Second, we do not actually care about which dish belongs to which equivalence class. The answer for the state "we still cannot tell dish 1 from dish 2 and we still cannot tell the dishes 3, 4, 5 apart" is clearly the same as the answer for the state where dishes 1 and 4 are still equivalent and dishes 2, 3, and 5 are still equivalent. Once we get rid of this symmetry, we see that our states correspond to *integer* *partitions*: different ways of writing a given positive integer (in our case, d) as a sum of positive integers given in non-descending order.

Finally, there is one technicality we need to pay attention to: one of our equivalence classes is actually special. This is the equivalence class of all dishes that have never been ordered. We need to keep track of its size separately, and we need to make sure that in the final state this class is empty.

Hard subproblem

One good thing to know about integer partitions: even though their number grows exponentially, the base of that exponential function is quite small, and therefore the partition numbers are surprisingly small as well. With 20 dishes, and even with the need to keep one of the partitions separate from the others, we are only looking at a few thousand states.

One slight issue is the number of transitions: the number of possible orders is p^d, so we need to bring that under control as well. How can we do that?

One trick that can be used in similar situations: increase the number of states! Instead of constructing and evaluating the entire order for a day at once, we can take the old state and process it incrementally: one equivalence class of dishes at a time. When processing an equivalence class of dishes, we decide how many of the people we have will order dishes from this class, and then we try all distinct possibilites – which can now, again, be seen as integer partitions. (This time, we are partitioning the people we have into groups that each orders a different dish. Hence, we are only interested in partitions that have at most as many elements as the number of dishes in our current equivalence class.)

Hence, in the new solution the general state will consist of four parts: some equivalence classes after the current day (those we already processed), some equivalence classes before the current day (waiting to be processed), the one special equivalence class (dishes that weren't ever ordered; this one is also waiting to be processed), and the number of people who still have to make their order for this day.

This change will increase the number of states to about 150 000 for the largest test case, but now we know that the number of transitions from each state is upper-bounded by the number of integer partitions of people, and that makes the solution fast enough.

Problem M: Make*me-an+[integer!]

Esoteric languages are a popular subject of IPSC problems. The home page lists many cases where we gave you an unusual language with strange syntax and even stranger semantics and tasked you with doing something useful in it.

But then we realized: why bother with esoteric languages? Why don't we just use a standard language everyone already knows? After all, that won't make it any easier for you!

Problem specification

In this problem, you will use JavaScript (as standardized by ECMA-262). All you have to do is produce the integers from 0 to 1000. There's just one catch: your programs can only use the characters `![]+-*`.

Input specification

There is no input.

Output specification

The output should contain exactly 1001 lines. For each i between 0 and 1000, line $i+1$ of your output should contain a valid JavaScript expression consisting only of the characters `![]+-*` that evaluates to a number (i.e., `typeof(result) == "number"`) with value i. Note that the expression must not contain any whitespace.

Additionally, your expressions must be short enough. For the **easy subproblem M1**, each JavaScript expression should be no longer than 200 characters. For the **hard subproblem M2**, no expression should exceed 75 characters.

Example output

```
+![]
+!![]
... (999 more lines)
```

Hints

Testing your solution: open your browser's developer console, or install node.js.

Pro tip: JavaScript seems like an easy language to learn. And it is, except for all its quirks and weird defaults. Unless you can score at least 21 out of 20 on quizzes such as this one, you shouldn't just assume some expression will throw a parse error or runtime error. Even strange expressions can be valid.

Task authors

Problemsetter:	Michal 'Mimino' Danilák, Peter 'PPershing' Perešíni
Task preparation:	Peter 'PPershing' Perešíni
Quality assurance:	Michal 'Mimino' Danilák

Solution

Easy subproblem

We will start by producing all the digits. Expressions for 0 and 1 were already showed in the example output: `+![]` and `+!![]`. In general, as soon as we can construct 1, we can simply add several 1s together. So, 2 is `+!![]+!![]` (or simply `!![]+!![]` as the initial cast of a boolean to an integer is no longer needed); 3 is `!![]+!![]+!![]`, and by now the idea should be clear.

The next step is producing multi-digit numbers. Here the big trick is string concatenation – in Javascript, concatenating two strings can be done using the `+` operator. In fact, `+` operator will also perform string concatenation if one argument is a string and the other is an integer. For example, `"1"+2+3` is `"123"`. But how to create the first string? Well, arrays (or more generally objects) are a good candidate, doing `[]+1` results in `"1"`. Or similarly, `[]+[] is ""`. Don't ask why.

Now we have only two problems left: The first one is that using `+` for string concatenation competes with `+` in our digits and we would end up just concatenating many 1s. Therefore, it would be nice to put the digit sub-expressions into parentheses. Unfortunately, characters `()` were not allowed in the problem statement. The trick is to use arrays – arrays have a high enough precedence and we can do something like `[expression][0]` to emulate `(expression)`.

Now we know how to concatenate the digits into a string. The final step is converting this string back into a number. A unary plus is our friend here: `+"123"` is 123. So, to put all things together, this is one way of writing 123: `+[[]+[+!![]][+![]]+[!![]+!![]][+![]]+[!![]+!![]+!![]][+![]]][+![]]`

Hard subproblem

The hard subproblem requires more work. We can start by systematically observing what our characters can do. After some experimenting with the javascript console, you could come up with the following set of observations:

- `!` always produces a boolean, no matter what value it is applied to.
- binary `*` and `-` first convert both arguments to a number and then compute the operation which results in a number (or NaN in case the conversion fails, or +infinity or -infinity in some very special cases)
- binary `+` is a chameleon. When at least one argument is a string, it concatenates strings. If both arguments are numbers and/or booleans, the result is a number.
- unary `+` and `-` always produce a number (or NaN). They can be used to convert string to a number
- as already shown, arrays can be used as parentheses, e.g., `(x)` is equivalent to `[x][0]`
- because we have no comma, we cannot create arrays with more than one element. Arrays with a single element are quite interesting though because they can act as strings: `[5]+1` is `51` but also `[5]*[5]` is 25.

We are able to construct expressions of various types. Here's a list of those:

- **a boolean:** could be used in binary `*`, `-`, and (numerical) `+`
- **a number(-like):** an expression which results in a number and behaves as a number regarding to the operator precedence
- **a numeric expression:** an expression which results a number but does not necessarily follow operator precedence. In particular, any expression which contains binary `+` or `-` is of this type.
- **a string(-like):** an expression which results in a string, or behaves like a string (e.g. `[x]`)
- **a string expression:** similar to a numeric expression but the result is a string. Again, no operator precedence is guaranteed.

We can formulate the following conversion rules between the types:

- Cast to a number: `+boolean/string/number` => `number`
- Cast to a number: `-boolean/string/number` => `number`
- Cast to a string: `[number/numeric expression/string/string expression]` => `string`
- "Downcast": `number/string` => `numeric expression/string expression`

and we can do the following operations:

- multiplication: `boolean/number/string * boolean/number/string` => `number`
- addition: `boolean/number/numeric expression + boolean/number/numeric expression` => `numeric expression`
- string concatenation: `number/numerical expression + string/string expression` => `string expression` (Note: precedence is left to right so numeric expression would be evaluated first)
- string concatenation: `string/string expression + number` => `string expression` (Note: cannot concatenate with numeric expression as it would break it into several terms concatenated together)
- subtraction: `boolean/number/numeric expression/string/string expression - boolean/number/string` => `numeric expression`

Given all these rules, we can now easily generate a solution for the hard input: We will start with the two booleans `![]` and `!![]` and we will follow all the conversions/operations possible while capping our search to reasonable numbers. (E.g., if the absolute value of the result is > 1100, it probably won't lead to a short result for numbers 0..1000 and we can cut the search there.)

There is one pitfall along the way: JavaScript does not like `++` and `--`. These are parsed as a special increment/decrement operator which can only be applied to a variable. Depending on the exact order in which you perform the conversion/operator rules, this might be a problem for the generated result. Our solution was to introduce two additional types: `+number` and `-number` which represent a number that might start with `+` or `-` respectively. We then disallow all operations that could lead to `++` and `--`.

But wait, there's more!

Using the set of rules given above, our solution can solve each of the given test cases in at most 66 characters. Still, it should be noted that there are other ways of producing numbers, and they can even be more efficient. We will give you one tasty example. We can create the number 100000000000000000000 simply as `+[1+"e"+2+0]`. OK, that would be neat, but where do we get an `"e"`? Easy: `"true"[3]` :) Thus, 10^{20} can be written as `+[+!![]+[!![]+[]][+![]][!![]+!![]+!![]]+[!![]+!![]]+[+![]]]`.