**1.Problem code: FCTRL**

The most important part of a GSM network is so called *Base Transceiver Station* (*BTS*). These transceivers form the areas called *cells* (this term gave the name to the cellular phone) and every phone connects to the BTS with the strongest signal (in a little simplified view). Of course, BTSes need some attention and technicians need to check their function periodically.

ACM technicians faced a very interesting problem recently. Given a set of BTSes to visit, they needed to find the shortest path to visit all of the given points and return back to the central company building. Programmers have spent several months studying this problem but with no results. They were unable to find the solution fast enough. After a long time, one of the programmers found this problem in a conference article. Unfortunately, he found that the problem is so called "Travelling Salesman Problem" and it is very hard to solve. If we have *N* BTSes to be visited, we can visit them in any order, giving us *N*! possibilities to examine. The function expressing that number is called factorial and can be computed as a product 1.2.3.4....*N*. The number is very high even for a relatively small *N*.

The programmers understood they had no chance to solve the problem. But because they have already received the research grant from the government, they needed to continue with their studies and produce at least *some* results. So they started to study behaviour of the factorial function.

For example, they defined the function *Z*. For any positive integer *N*, *Z*(*N*) is the number of zeros at the end of the decimal form of number *N*!. They noticed that this function never decreases. If we have two numbers *N* 1 <*N* 2 , then *Z*(*N* 1 ) <= *Z*(*N* 2 ). It is because we can never "lose" any trailing zero by multiplying by any positive number. We can only get new and new zeros. The function *Z* is very interesting, so we need a computer program that can determine its value efficiently.

**Input**

There is a single positive integer *T* on the first line of input (equal to about 100000). It stands for the number of numbers to follow. Then there are *T* lines, each containing exactly one positive integer number *N*, 1 <= *N* <= 1000000000.

**Output**

For every number *N*, output a single line containing the single non-negative integer *Z*(*N*).

**Example**

Sample Input:

6

3

60

100

1024

23456

8735373

Sample Output:

0

14

24

253

5861

2183837

2. **Problem code: MMIND**

If you want to buy a new cellular phone, there are many various types to choose from. To decide which one is the best for you, you have to consider several important things: its size and weight, battery capacity, WAP support, colour, price. One of the most important things is also the list of games the phone provides. Nokia is one of the most successful phone makers because of its famous Snake and Snake II. ACM wants to make and sell its own phone and they need to program several games for it. One of them is Master-Mind, the famous board logical game.

The game is played between two players. One of them chooses a *secret code* consisting of *P* ordered pins, each of them having one of the predefined set of *C* colours. The goal of the second player is to guess that secret sequence of colours. Some colours may not appear in the code, some colours may appear more than once.

The player makes guesses, which are formed in the same way as the secret code. After each guess, he/she is provided with an information on how successful the guess was. This feedback is called a *hint*. Each hint consists of *B* black points and *W* white points. The black point stands for every pin that was guessed right, i.e. the right colour was put on the right position. The white point means right colour but on the wrong position. For example, if the secret code is "white, yellow, red, blue, white" and the guess was "white, red, white, white, blue", the hint would consist of one black point (for the white on the first position) and three white points (for the other white, red and blue colours). The goal is to guess the sequence with the minimal number of hints.

The new ACM phone should have the possibility to play both roles. It can make the secret code and give hints, but it can also make its own guesses. Your goal is to write a program for the latter case, that means a program that makes Master-Mind guesses.

**Input**

There is a single positive integer *T* on the first line of input. It stands for the number of test cases to follow. Each test case describes one game situation and you are to make a guess. On the first line of each test case, there are three integer numbers, *P*, *C* and *M*. *P* ( 1 <= *P* <= 10) is the number of pins, *C* (1 <= *C* <= 100) is the number of colours, and *M* (1 <= *M* <= 100) is the number of already played guesses.

Then there are 2 *x M* lines, two lines for every guess. At the first line of each guess, there are *P* integer numbers representing colours of the guess. Each colour is represented by a number *G i* , 1 <= *G i* <= *C*. The second line contains two integer numbers, *B* and *W*, stating for the number of black and white points given by the corresponding hint.

Let’s have a secret code *S* 1 , *S* 2 , ... *S P* and the guess *G* 1 , *G* 2 , ... *G P* . Then we can make a set *H* containing pairs of numbers (*I*,*J*) such that *S I* = *G J* , and that any number can appear at most once on the first position and at most once on the second position. That means for every two different pairs from that set, (*I* 1 ,*J* 1 ) and (*I* 2 ,*J* 2 ), we have *I* 1 <> *I* 2 and *J* 1 <> *J* 2 . Then we denote *B*(*H*) the number of pairs in the set, that meet the condition *I* = *J*, and *W*(*H*) the number of pairs with *I* <> *J*.

We define an ordering of every two possible sets *H* 1 and *H* 2 . Let’s say *H* 1 <= *H* 2 if and only if one of the following holds:

*B*(*H* 1 ) < *B*(*H* 2 ), or

*B*(*H* 1 ) = *B*(*H* 2 ) and *W*(*H* 1 ) <= *W*(*H* 2 )

Then we can find a maximal set *H max* according to this ordering. The numbers *B*(*H max* ) and *W*(*H max* )

are the black and white points for that hint.

**Output**

For every test case, print the line containing *P* numbers representing *P* colours of the next guess. Your guess must be valid according to all previous guesses and hints. The guess is valid if the sequence could be a secret code, i.e. the sequence was not eliminated by previous guesses and hints.

If there is no valid guess possible, output the sentence You are cheating!. If there are more valid guesses, output the one that is lexicographically smallest. I.e. find such guess *G* that for every other valid guess *V* there exists such a number *I* that:

*G J* = *V J* for every *J*<*I*, and

*G I* <*V I* .

**Example**

Sample Input:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 |  |  |  |  | | | |
| 4 | 3 | 2 |  |
| 1 | 2 | 3 | 2 |
| 1 | 1 |  |  |
| 154 | 5 | 7 |  |
| 2 | 1 | 3 | 2 |
| 1 | 1 |  |  |
| 4 | 6 | 2 |  |
| 3 | 3 | 3 | 3 |
| 3 | 0 |  |  |
| 4 | 4 | 4 | 4 |
| 2 | 0 |  |  |
| 8 | 9 | 3 |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0 | 0 |  |  |  |  |  |  |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 0 |  |  |  |  |  |  |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 | 9 |
| 2 | 0 |  |  |  |  |  |  |

Sample Output

1 1 1 3

You are cheating!

9 9 9 9 9 9 9 9

3. **Problem code: HOTLINE**

Every customer sometimes needs help with new and unusual products. Therefore, hotline service is very important for every company. We need a single phone number where the customer can always find a friendly voice ready to help with anything. On the other hand, many people are needed to serve as hotline operators, and human resources are always very expensive. Moreover, it is not easy to pretend "friendly voice" at 4am and explain to a drunken man that you are really unable to give him the number to House of Parliament. It was also found that some of the questions repeat very often and it is very annoying to answer them again and again.

ACM is a modern company, wanting to solve its hotline problem. They want to decrease the number of human operators by creating a complex software system that would be able to answer most common questions. The customer’s voice is analysed by a special Voice Recognition Module (VRM) and converted to a plain text. The text is then taken by an Artificial Automatic Adaptive Answering Algorithm (AAAAA). The most common questions are recognised and answered automatically. The replies are then converted to a sound by Text-to-Speech Module (TTS).

You are to write the AAAAA module. Because your algorithm should be adaptive, it has no explicit knowledge base. But it must be able to listen to sentences in English and remember the mentioned facts. Whenever the question is asked about such a fact, the system has to answer it properly. The VRM and TTS modules are already implemented, so the input and output of AAAAA will be in the text form.

**Input**

There is a single positive integer *T* on the first line of input. It stands for the number of dialogues to follow. Each dialogue consists of zero or more lines, each of them containing one sentence: either statement or question. The statement ends with a dot character (.), the question ends with a question mark (?). No statement will appear more than once, however the questions can be repeated. There is one extra line after each dialogue. That line ends with an exclamation mark (!).

Sentences can contain words, spaces and punctuation characters (such as commas, colons, semicolons etc.). All words contain only letters of English alphabet and are case-sensitive. That means the same word is always written the same way, usually in lowercase. Acronyms, names and some other words can begin with capital letters. For simplicity, all sentences begin with a lowercase letter. Only if the first word should be written with a capital, the sentence begins with a capital letter. There are no unneeded spaces between words. No line will have more than 100 characters. There will be at most

100 statements per each test case.

**Statements**

Each statement has one of the following two forms ( \_ denotes a space):

*subject* \_*predicate*[s] [ \_*object*] .

*subject* \_don’t|doesn’t \_*predicate* [ \_*object*] .

The square brackets mark an optional part, the vertical line two possible variants. Subject is a single word, noun or pronoun in singular. Predicate is a verb (single word) denoting some activity. Object can be any text. Object does not contain any dots. Any pair "verb + object" determines unique activity. The same verb with different objects makes different independent activities, i.e. the different and independent meaning of the sentence. Sentence without any object can be considered as sentence with an empty object. The verb without an object has different and independent meaning than the same verb with any non-empty object.

The first variant of sentence denotes a positive statement. The word "*predicate*[s]" means verb that matches the subject of the sentence. If the subject is "I" or "you", the verb has the same form as the infinitive. With any other subject, the letter "s" is appended on the end of the verb. Assume there are no irregular verbs.

The second variant is a negative statement. Verb "don’t" or "doesn’t" must also match the subject. The form "don’t" is used with either "I" or "you", "doesn’t" is used in any other case.

A special generic subject "everybody" can be used. It means the activity holds for any subject. Other special subject is "nobody". Such sentence also holds for any subject, but its meaning is negative. Both of these generic subjects can be used with the first variant only (without "doesn’t"). The sentence "nobody likes something" is exactly equal to "everybody doesn’t like something", except the latter form will never occur in the input.

**Questions**

Each question has one of the following three forms:

1. do|does \_*subject* \_*predicate* [ \_object] ?

2. who \_*predicate*s [ \_*object*] ?

3. what \_do|does \_*subject* do ?

The word "do|does" always matches the subject ("do I?", "do you?", "does any other subject?"). In the second type of question, predicate always matches the word "who", i.e. the "s" is always appended. Generic subjects cannot be used in questions.

**Output**

For each dialogue, your program must output the line Dialogue #*D*:, where *D* is the sequence number of dialogue, starting with 1. Then print exactly three lines for every question: the first line repeats the question, the second line contains the answer, and the third line is empty. Print nothing for statements. After each dialogue, print the same line with an exclamation mark that was in the input. Then print one extra empty line. Empty line contains a new-line character only.

The answer must be properly formated to be accepted by a TTS module. Only the statements appearing in the input before the answer are used for the corresponding reply. If there is any contradiction among statements, the reply is always I am abroad.. If the question and statements consider the special subject "you", it must be replaced with "I" in the answer. If the question considers special subject "I", it must be replaced with "you" in the answer. The verb must always match the subject of the sentence. The exact form of the correct answer depends on the type of question.

**1. does subject predicate [object] ?**

If there is any positive statement about the mentioned subject (or generic subject "everybody"), predicate and object, the answer is:

yes, \_*subject* \_*predicate*[s] [ \_*object*] .

If there is any negative statement about the mentioned subject (or generic subject "nobody"), predicate and object, the answer is:

no, \_*subject* \_don’t|doesn’t \_*predicate* [ \_*object*] .

Otherwise, the answer is: maybe.

Subject in the answer is always the same subject as the subject of the question.

**2. who predicates [object] ?**

If there is a positive statement considering any subject, the specified predicate and object, the answer is:

*subject* \_*predicate*[s] [ \_*object*] .

If two or more subjects match the activity, replace the subject in the answer with enumeration of all such subjects, in the same order as the corresponding statements have appeared in the input. Subjects are separated with comma and space, last two subjects are separated with the word "and". If "everybody" belongs to the group of enumerated subjects, do not enumerate subjects, and print "everybody" only. If the enumeration contains at least two subjects, the predicate matches the plural subject (i.e. verb is without trailing "s"), otherwise it matches the only subject.

*subject1* , \_*subject2* \_and \_*subject3 predicate* [ \_*object*] .

If there is a negative statement considering the generic subject "nobody", the specified predicate and object, the answer is:

*nobody* \_*predicate*s [ \_*object*] .

Otherwise, the answer is: I don’t know.

**3. what does subject do ?**

If there are one or more sentences (both positive and negative) considering the specified subject (or a generic subject "everybody" or "nobody"), all verbs and objects from such sentences must be included in a reply in the same order as the corresponding sentences have appeared in the input. No verb-object pair can be included more than once (the eventual second appearance must be skipped). The verb-object pairs are separated by a comma followed by a space, the last verb is separated by a comma and the word "and". Please note the comma is printed here although there was no comma when separating the subjects in the previous type of answer (see above). The negative answers have the same form as the statements, that means the verb "don’t" or "doesn’t" is used:

*subject* [ \_don’t|doesn’t] \_*predicate1*[s] [ \_*object1*] ,

[ \_don’t|doesn’t] \_*predicate2*[s] [ \_*object2*] ,

\_and [ \_don’t|doesn’t] \_*predicate3*[s] [ \_*object3*] .

*subject* [ \_don’t|doesn’t] \_*predicate1*[s] [ \_*object1*] ,

\_and [ \_don’t|doesn’t] \_*predicate2*[s] [ \_*object2*] .

*subject* [ \_don’t|doesn’t] \_*predicate*[s] [ \_*object*] .

Otherwise, the answer is: I don’t know.

**Example**

Sample Input:

1

I like hotdogs.

nobody likes to work. everybody smiles.

what do I do?

who smiles?

what do you do?

does Joe smile?

do I like to work? everybody hurts sometimes. who walks there?

Michal walks there. who walks there?

what does Michal do? do you understand? nobody walks there.

do you understand now?

bye!

Sample Output:

Dialogue #1:

what do I do?

you like hotdogs, don’t like to work, and smile.

who smiles?

everybody smiles.

what do you do?

I don’t like to work, and smile.

does Joe smile?

yes, Joe smiles.

do I like to work?

no, you don’t like to work.

who walks there? I don’t know.

who walks there? Michal walks there.

what does Michal do?

Michal doesn’t like to work, smiles, hurts sometimes, and walks there.

do you understand?

maybe.

do you understand now? I am abroad.

bye!

**4.Problem code: IKEYB**

Most of you have probably tried to type an SMS message on the keypad of a cellular phone. It is sometimes very annoying to write longer messages, because one key must be usually pressed several times to produce a single letter. It is due to a low number of keys on the keypad. Typical phone has twelve keys only (and maybe some other control keys that are not used for typing). Moreover, only eight keys are used for typing 26 letters of an English alphabet. The standard assignment of letters on the keypad is shown in the left picture:

|  |  |  |
| --- | --- | --- |
| 1 | 2 abcd | 3 efg |
| 4 hijk | 5 lm | 6 nopq |
| 7 rs | 8 tuv | 9 wxyz |
| \* | 0  *space* | # |

|  |  |  |
| --- | --- | --- |
| 1 | 2 abc | 3 def |
| 4 ghi | 5 jkl | 6 mno |
| 7 pqrs | 8 tuv | 9 wxyz |
| \* | 0  *space* | # |

There are 3 or 4 letters assigned to each key. If you want the first letter of any group, you press that key once. If you want the second letter, you have to press the key twice. For other letters, the key must be pressed three or four times. The authors of the keyboard did not try to optimise the layout for minimal number of keystrokes. Instead, they preferred the even distribution of letters among the keys. Unfortunately, some letters are more frequent than others. Some of these frequent letters are placed on the third or even fourth place on the standard keyboard. For example, S is a very common letter in an English alphabet, and we need four keystrokes to type it. If the assignment of characters was like in the right picture, the keyboard would be much more comfortable for typing average English texts.

ACM have decided to put an optimised version of the keyboard on its new cellular phone. Now they need a computer program that will find an optimal layout for the given letter frequency. We need to preserve alphabetical order of letters, because the user would be confused if the letters were mixed. But we can assign any number of letters to a single key.

**Input**

There is a single positive integer *T* on the first line of input (equal to about 2000). It stands for the number of test cases to follow. Each test case begins with a line containing two integers *K*, *L* (1 <= *K*

<= *L* <= 90) separated by a single space. *K* is the number of keys, *L* is the number of letters to be mapped onto those keys. Then there are two lines. The first one contains exactly *K* characters each representing a name of one key. The second line contains exactly *L* characters representing names of letters of an alphabet. Keys and letters are represented by digits, letters (which are case-sensitive), or any punctuation characters (ASCII code between 33 and 126 inclusively). No two keys have the same character, no two letters are the same. However, the name of a letter can be used also as a name for a key.

After those two lines, there are exactly *L* lines each containing exactly one positive integer *F* 1 , *F* 2 , ... *F L* . These numbers determine the frequency of every letter, starting with the first one and continuing with the others sequentially. The higher number means the more common letter. No frequency will be higher than 100000.

**Output**

Find an optimal keyboard for each test case. Optimal keyboard is such that has the lowest "price" for typing average text. The *price* is determined as the sum of the prices of each letter. The price of a letter is a product of the letter frequency (*F i* ) and its position on the key. The order of letters cannot be changed, they must be grouped in the given order.

If there are more solutions with the same price, we will try to maximise the number of letters assigned to the last key, then to the one before the last one etc.

More formally, you are to find a sequence *P* 1 , *P* 2 , ... *P L* representing the position of every letter on a particular key. The sequence must meet following conditions:

*P* 1 = 1

for each *i*>1, either *P i* = *P i*-1 +1 or *P i* = 1

there are at most *K* numbers *P i* such that *P i* = 1

the sum of products *S P* = Sum[i=1..l] *F i* .*P i* is minimal

for any other sequence *Q* meeting these criteria and with the same sum *S Q* = *S P* , there exists such

*M*, 1 <= *M* <= *L* that for any *J*, *M*<*J* <= *L*, *P J* = *Q J* , and *P M* >*Q M* .

The output for every test case must start with a single line saying Keypad #*I*:, where *I* is a sequential order of the test case, starting with 1. Then there must be exactly *K* lines, each representing one letter, in the same order that was used in input. Each line must contain the character representing the key, a colon, one space and a list of letters assigned to that particular key. Letters are not separated from each other.

Print one blank line after each test case, including the last one.

**Example**

Sample Input:

1

8 26

23456789

ABCDEFGHIJKLMNOPQRSTUVWXYZ

3371

589

1575

1614

6212

971

773

1904

2989

123

209

1588

1513

2996

3269

1080

121

2726

3083

4368

1334

518

752

427

733

871

Sample Output:

Keypad #1:

2: ABCD

3: EFG

4: HIJK

5: LM

6: NOPQ

7: RS

8: TUV

9: WXYZ

**5.Problem code: SHPATH**

Given a list of cities. Each direct connection between two cities has its transportation cost (an integer bigger than 0). The goal is to find the paths of minimum cost between pairs of cities. Assume that the cost of each path (which is the sum of costs of all direct connections belongning to this path) is at most

200000. The name of a city is a string containing characters a,...,z and is at most 10 characters long.

**Input**

*s* [the number of tests <= 10]

*n* [the number of cities <= 10000]

*NAME* [city name]

*p* [the number of neighbours of city *NAME*]

*nr cost* [*nr* - index of a city connected to *NAME* (the index of the first city is 1)] [*cost* - the transportation cost]

*r* [the number of paths to find <= **100**]

*NAME1 NAME2* [*NAME1* - source, *NAME2* - destination] [empty line separating the tests]

**Output**

*cost* [the minimum transportation cost from city *NAME1* to city *NAME2* (one per line)]

**Example**

Input:

1

4 gdansk

2

2 1

3 3 bydgoszcz

3

1 1

3 1

4 4 torun

3

1 3

2 1

4 1 warszawa

2

2 4

3 1

2

gdansk warszawa bydgoszcz warszawa

Output:

3

2