Have you tried this horrible-looking puzzle?

1. Which question is the first question whose answer is b? (a) 2; (b) 3; (c) 4; (d) 5; (e) 6;
2. The only question that has the same answer as its next question is (e.g. option e means question 6 and 7's answers are the same): (a) 2; (b) 3; (c) 4; (d) 5; (e) 6;
3. Among the 5 options, which question has the same answer as this question (i.e. question 3)? (a) 1; (b) 2; (c) 4; (d) 7; (e) 6;
4. How many questions' answer is a? (a) 0; (b) 1; (c) 2; (d) 3; (e) 4
5. Which of the following questions has the same answer as this question? (a) 10; (b) 9; (c) 8; (d) 7; (e) 6;
6. The number of questions whose answer is a, equals the number of questions whose answer is: (a) b; (b) c; (c) d; (d) e; (e) none of above
7. What is the difference of this question's answer and the next question's answer (e.g. the difference of a and b is 1) ? (a) 4; (b) 3; (c) 2; (d) 1; (e) 0;
8. How many questions' answer is a vowel? (only a and e are vowels. Others are consonants) (a) 2; (b) 3; (c) 4; (d) 5; (e) 6
9. The number of questions whose answer is a consonant is: (a) a prime; (b) a factorial; (c) a square number; (d) a cubic number; (e) a multiple of 5
10. The answer of this question is: (a) a; (b) b; (c) c; (d) d; (e) e;

Note:

1. make sure that your answer is not self-contradicting. For example, the first question's answer can't be b.
2. make sure that for each question, only your answer is correct, all other options must be incorrect. For example, if your answer to question 5 is a, then none of question 9, 8, 7, 6's answers can be a!
3. make sure that your answer won't make any question invalid. For example, if question 2 and 3's answer are the same, and question 8,9's answers are also the same, question 2 would be invalid (because no question is "the only question" that satisfying the condition)

It's possible to solve this problem by hand, but as a programmer, solving it with a program is more fun!

How to Solve the Puzzle with a Program

Here's one way: enumerate all possible answers ($5^{10} = 9765625$), and for each question, check whether only your answer is correct. Pseudo-code:

```
forall(answer_list):
  bad = False
  for testing_question in [1,2,3,4,5,6,7,8,9,10]:
    for testing_option in ["a","b","c","d","e"]:
      # your answer should be correct
      if testing_option == answer_list[testing_question] and
check(testing_question, testing_option) == False:
        bad = True
      # other options must be incorrect
      if testing_option != answer_list[testing_question] and
check(testing_question, testing_option) == True:
        bad = True
  if not bad:
    print answer_list
```

Here "`answer_list`" is a list of letters (subscript is 1-based), where the i-th letter is the answer to the i-th question.

Believe or not, the *only* answer is: "`cdebeedcba`" (if you prefer to add the question index before each answer, it is "`1c2d3e4b5e6e7d8c9b10a`")

Amazing, huh? There's more. You wish that your program could solve some other puzzles, but first of all, you need to formulate the puzzle in a formal language.

Formalizing the Puzzle

This problem uses a LISP dialect to represent the puzzle. Don't worry if you don't know LISP, it has a very simple syntax. `(f a b)` means calling a function f with parameters a and b. That's like `f(a, b)` in C/C++/Java. Similarly, `(f a (g b c) d)` is like `f(a, g(b, c), d)` in C/C++/Java.

Here is an example of how to describe a question of the puzzle:

```
3. (equal (answer 3) (answer (option-value)))
a. 1
b. 2
c. 4
d. 7
e. 6
```

There are two very important built-in functions involved:

`(answer idx)`	returns $answer\_list[idx]$ in the pseudo-code above.
`(option-value)`	returns the "evaluation result" of testing_option's text, treated as an expression.

In the example above, if testing_option is "c", then (option-value) returns 4 (an integer) because 4 is the expression presented in the option "c" of this question. Note that testing_option's text can be a complex expression instead of a simple value. Refer to Sample Input.

The function "`check(testing_question, testing_option)`" above can be implemented as follows:

```
check(testing\_question, testing\_option):
  1. set-up the function (option-value) so that it returns the evaluation
result of testing\_option of testing\_question
  2. evaluate the lisp expression of testing\_question (e.g. the expression
(equal (answer 3) (answer (option-value))) in the example above)
  3. if an unhandled exception is raised during the evaluation, returns False
  4. if the result of step 2 is boolean, return it; otherwise return False
```

There is one special option expression: "none-of-above". The result of "none-of-above" depends on other options' evaluation results. In this problem, there can be at most one "none-of-above" for each question, and it must be the last option.

Details

Here are the details of the LISP dialect used in this problem:

- There are four datatypes: integer, string, boolean and functions.
- There are only two boolean values: true and false. Note that there are no "boolean literal", so you don't care whether to use `#t` and `#f` (like in Scheme), or `t` and nil (like in Common Lisp) to represent boolean constants.
- Integers are always non-negative integers.
- String literals are always enclosed by double quotes, like "`a string`".
- There is no variable. All the so-called "`identifiers`" (consisting of letters and hyphens) are always pre-defined functions.

Below is a list of pre-defined functions. Functions starting with '`!`' means it may throw an exception, and functions starting with '`@`' means it can handle exceptions. Like C++/Java/Python, once an exception is raised, the evaluation process is stopped unless a function handles the exception. In the text below, '`iff`' means "if and only if".

Basic functions

`(equal a b)`	return true iff a and b are of the same type and are equal. In this problem, you'll never need to compare two functions.
`(option-value)`	iscussed above.
`!(answer idx)`	discussed above. If idx is not an integer or is not in $1 \dots n$ (where n is the number of questions), then raises an exception
`!(answer-value idx)`	Returns the "evaluation result" of option answer_list[idx] of question idx. Also raises an exception on error.

Predicates

Predicate is a special kind of function. It always takes a value of any type and returns a boolean value.

`prime-p`	returns true iff the parameter is a positive prime
`factorial-p`	self-explanatory
`square-p`	self-explanatory
`cubic-p`	self-explanatory
`vowel-p`	returns true iff the parameter is a single letter and is a vowel
`consonant-p`	self-explanatory

Queries and statistics

`!@(first-question pred)`	Return id of the first question that satisfies 'pred'. Raises an exception if not found.
`!@(last-question pred)`	Return id of the last question that satisfies 'pred'. Raises an exception if not found.
`!@(only-question pred)`	Return id of the only question that satisfies 'pred'. Raises an exception if not found or more than one question found.
`@(count-question pred)`	Return the number of questions that satisfies 'pred'.
`!(diff-answer idx1 idx2)`	The difference of answers of question idx1 and idx2. Raises an exception on error, otherwise the return value is always $0 \dots m-1$, where m is the number of options for each question.

Note that in the first four functions (those with a '`@`' flag), if an exception was raised when evaluating the predicate, the exception is handled and the predicate is not considered satisfied. For example, if answer_list is "`abc`", `(count-question (make-answer-diff-next-equal 0))` returns 0 and doesn't raise an exception, even though evaluating the predicate for question 3, i.e. `((make-answerdiff-next-equa 0) 3)` raised an exception. However, all other functions will not handle exceptions. For example, if there are only 3 questions, `(factorial-p (answer-value 5))` will raise an exception instead of returning false.

Predicate generators

There are also functions that can create predicates on-the-fly:

`!(make-answer-diff-next- equal num)`	returns a predicate (p idx) which evaluates (diff-answer idx idx+1) and returns true if the result equals to num. Raises an exception if num is not an integer.
`(make-answer-equal a)`	returns a predicate (p idx) which evaluates (answer idx) and returns true if the result equals a.
`(make-answer-is pred)`	returns a predicate (p idx) which evaluates (answer idx) and returns true if the result satisfies 'pred'.
`(make-answer-value-equal a)`	self-explanatory. The predicate evaluates (answer-value idx)
`(make-answer-value-is pred)`	self-explanatory. The predicate evaluates (answer-value idx)
`!(make-is-multiple num)`	returns a predicate (p i) which returns true iff i is an integer and is a multiple of num. Raises an exception if num is not an integer.
`!(make-equal val)`	returns a predicate (p v) which returns true iff (equal v val) is true. Raises an exception if val is neither an integer nor a string.
`(make-not pred)`	returns a predicate (p v) which returns true iff (pred v) is false.
`(make-and pred1 pred2)`	returns a predicate (p v) which returns true iff (pred1 v) and (pred2 v) are both true. Both pred1 and pred2 need to be evaluated. No short-circuit operation should be done.
`(make-or pred1 pred2)`	returns a predicate (p v) which returns true iff at least one of (pred1 v) and (pred2 v) is true. Both pred1 and pred2 need to be evaluated. No short-circuit operation should be done.

For example, `(make-is-multiple 3)` returns a predicate "is a multiple of 3", so `((make-is-multiple 3) 6)` returns true and `((make-is-multiple 3) 10)` returns false. Similarly `(make-not (make-or square-p prime-p))` returns a predicate "neither a square nor a prime".

Input

There will be at most 50 test cases. Each test case begins with two integers n and m ($2 \leq n \leq 10$, $2 \leq m \leq 5$), the number of questions and the number of options per question. Each question is described with $m + 1$ lines: the question's expression and the options. Questions are numbered $1 \dots n$, and options are labeled '`a`'..'`e`'. Options are valid expressions and will not call option-value (calling optionvalue makes it recursive!). Each question is followed by a blank line. Most test cases are easy.

Output

For each test case, print the case number in the first line, and a list of answers, one per line, sorted in ascending order. There will always be at least one answer.

Note: This problem is complex. If you have any questions, email me: `rujia.liu@gmail.com`

Sample Input

```
3 3
(equal (option-value) (count-question (make-answer-equal "a")))
3
0
1

(equal (option-value) "a")
"c"
"b"
"a"

((option-value) (count-question (make-answer-equal "c")))
(make-and (make-is-multiple 2) (make-or factorial-p prime-p))
(make-not prime-p)
"none-of-above"

3 2
(equal (option-value) (answer 2))
"a"
"none-of-above"

(equal (option-value) (first-question (make-answer-diff-next-equal 0)))
1
2

((option-value) (last-question (make-answer-equal "b")))
(make-is-multiple 2)
(make-not (make-is-multiple 2))

3 2
(equal (option-value) (answer 1))
"a"
"b"

((option-value) (last-question (make-answer-diff-next-equal 0)))
(make-equal 2)
"none-of-above"

((option-value) (only-question (make-answer-equal "b")))
(make-is-multiple 2)
"none-of-above"

2 5
((option-value) (diff-answer 1 2))
factorial-p
prime-p
(make-not square-p)
(make-not cubic-p)
"none-of-above"

(equal (only-question (option-value)) 1)
(make-answer-is consonant-p)
(make-answer-is vowel-p)
(make-answer-value-equal 1)
(make-answer-value-is square-p)
"none-of-above"

2 2
(option-value)
(equal (first-question (make-answer-diffnext-equal 2)) (first-question (makeanswer-diff-next-equal 2)))
"none-of-above"

(equal (option-value) 1)
1
2
```

Sample Output

```
Case 1:
bcb
cca
Case 2:
aab
Case 3:
aba
Case 4:
ab
ee
Case 5:
ba
```