Colombian Collegiate Programming League
CCPL 2013

Contest 3 -- April 6

Strings

Problems

This set contains 8 problems; pages 1 to 18.

(Borrowed from several sources online.)

A - Sonnet Rhyme Verifier

Source file name: `sonnet.c`, `sonnet.cpp` *or* `sonnet.java`

According to Wikipedia

> The term *sonnet* derives from the Provencal word *sonet* and the Italian word *sonetto*, both meaning ''little song''. By the thirteenth century, it had come to signify a poem of fourteen lines that follows a strict rhyme scheme and logical structure. The conventions associated with the sonnet have evolved over its history. The writers of sonnets are known as *sonneteers*.

Written in Spanish, sonnets have a well defined structure over the rhymes. Moreover, these days sonneteers are scarce as good weather and the rules governing the rhymes are rather permissive. Let us consider the following poem:

```
ES ELLA
Locura, intensa y sin medida
andando errante te avisté,
una nota en tus manos encontré:
rumor de un amor y su partida.
Adelante, te diré de su vida
aunque en mi regazo ya no esté,
fueron sus labios con que tropecé
los que te llamaron despavorida.
Ojos negros, pestañas en un ramo,
rizos finos, humor, siempre bella;
¡es ella, amiga, a quien amo!
Zagales de secretos sus estrellas,
cómplices la lluvia y el álamo;
... es ella, nuestra musa, ¡es ella!
```

This poem must be consider as sonnet because it meets the following rules:

1. It has 14 lines.
2. Its rhymes have the structure $ABBAABBACDCDCD$, where A, B, C and D correspond to the suffixes `ida`, `é`, `amo`, and `ella` of the given line. In general, we are to allow any rhyme structure as long as it follows one of the patterns $ABBAABBACDECDE$, $ABBAABBACDEDCE$ or $ABBAABBACDCDCD$.
3. It has hendecasyllable meter.

Today you are to help the local Spanish School of Sonneteers (SSS) writing a program that verifies if a given composition sticks to the first two rules of a Spanish sonnet. The last rule is to be checked by the SSS's master sonneteer once the given poem passes the tests exercised by the verifier you are about to code.

For the purpose of your work, two lines rhyme if, after *cleaning* them, they have the same suffix. Given a line in the sonnet, its clean version is the result of deleting blanks, punctuation symbols (`¡`, `!`, `,`, `.`, `:`, `;`, `¿`, `?`, `-`) and the last s (if there is one) from the end of the original line.

Input

The input consists of several instances of the problem, each one occurring separated by a new line. Each instance consists of $n > 2$ lines: the first one consists of a list of s suffixes ($4 \leq s \leq 5$), separated by a blank character. The suffixes are to be called A, B, C, D and E according to the order in which they occur and the total number of them. The second line contains the title of the composition. The following $n - 2$ lines contain the actual lines of the composition. Each line in the sonnet has at most 80 characters. You can safely assume that, in a case, a given suffix is not suffix of another one.

The input must be read from standard input.

Output

For each problem instance, and according to the order of the input, your program should print a sentence of the form
`<NAME>: <STRUCTURE>`
if the given composition adheres to the first two rules of a spanish sonnet or
`<NAME>: Not a chance!`
in other case, where `<NAME>` is to be replaced by the name of the composition and `<STRUCTURE>` corresponds to the structure of the rhyme, and is given by the order in which the suffixes occur.

The output must be written to standard output.

Sample input	**Sample output**
ida é amo ella ES ELLA Locura, intensa y sin medida andando errante te avisté, una nota en tus manos encontré: rumor de un amor y su partida. Adelante, te diré de su vida aunque en mi regazo ya no esté, fueron sus labios con que tropecé los que te llamaron despavorida. Ojos negros, pestañas en un ramo, rizos finos, humor, siempre bella; ¡es ella, amiga, a quien amo! Zagales de secretos sus estrellas, cómplices la lluvia y el álamo; ... es ella, nuestra musa, ¡es ella! ura ima illa eto OTrO Esta composición posiblemente rima pero nunca será un soneto con esta frase ya no sirve ni tampoco porque está incompleto.	ES ELLA: ABBAABBACDCDCD OTrO: Not a chance!

B - GATTACA

Source file name: `gattaca.c`, `gattaca.cpp` *or* `gattaca.java`

The Institute of Bioinformatics and Medicine (IBM) of your country has been studying the DNA sequences of several organisms, including the human one. Before analyzing the DNA of an organism, the investigators must extract the DNA from the cells of the organism and decode it with a process called ''sequencing''.

A technique used to decode a DNA sequence is the ''shotgun sequencing''. This technique is a method applied to decode long DNA strands by cutting randomly many copies of the same strand to generate smaller fragments, which are sequenced reading the DNA bases (`A`, `C`, `G` and `T`) with a special machine, and re-assembled together using a special algorithm to build the entire sequence.

Normally, a DNA strand has many segments that repeat two or more times over the sequence (these segments are called ''*repetitions*''). The repetitions are not completely identified by the shotgun method because the re-assembling process is not able to differentiate two identical fragments that are substrings of two distinct repetitions.

The scientists of the institute decoded successfully the DNA sequences of numerous bacterias from the same family, with other method of sequencing (much more expensive than the shotgun process) that avoids the problem of repetitions. The biologists wonder if it was a waste of money the application of the other method because they believe there is not any large repeated fragment in the DNA of the bacterias of the family studied.

The biologists contacted you to write a program that, given a DNA strand, finds the largest substring that is repeated two or more times in the sequence.

Input

The first line of the input contains an integer T specifying the number of test cases ($1 \leq T \leq 100$). Each test case consists of a single line of text that represents a DNA sequence S of length n ($1 \leq n \leq 1000$).

You can suppose that each sequence S only contains the letters `A`, `C`, `G` and `T`.

The input must be read from standard input.

Output

For each sequence in the input, print a single line specifying the largest substring of S that appears two or more times repeated in S, followed by a space, and the number of ocurrences of the substring in S.

If there are two or more substrings of maximal length that are repeated, you must choose the least according to the lexicographic order. If there is no repetition in S, print ''`No repetitions found!`''

The output must be written to standard output.

Sample input	**Sample output**
6 GATTACA GAGAGAG GATTACAGATTACA TGAC TGTAC TTGGAACC	A 3 GAGAG 2 GATTACA 2 No repetitions found! T 2 A 2

C - Phylogenetic Trees Inherited

Source file name: `phylogen.c`, `phylogen.cpp` *or* `phylogen.java`

Among other things, Computational Molecular Biology deals with processing genetic sequences. Considering the evolutionary relationship of two sequences, we can say that they are closely related if they do not differ very much. We might represent the relationship by a tree, putting sequences from ancestors above sequences from their descendants. Such trees are called phylogenetic trees.

Whereas one task of phylogenetics is to infer a tree from given sequences, we'll simplify things a bit and provide a tree structure - this will be a complete binary tree. You'll be given the n leaves of the tree. Sure you know, n is always a power of 2. Each leaf is a sequence of amino acids (designated by the one-character-codes you can see in the figure . All sequences will be of equal length l. Your task is to derive the sequence of a common ancestor with minimal costs.

Amino Acid		
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic Acid	Asp	D
Cysteine	Cys	C
Glutamine	Gln	Q
Glutamic Acid	Glu	E
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

The costs are determined as follows: every inner node of the tree is marked with a sequence of length l, the cost of an edge of the tree is the number of positions at which the two sequences at the ends of the edge differ, the total cost is the sum of the costs at all edges. The sequence of a common ancestor of all sequences is then found at the root of the tree. An optimal common ancestor is a common ancestor with minimal total costs.

Input

The input file contains several test cases. Each test case starts with two integers n and l, denoting the number of sequences at the leaves and their length, respectively. Input is terminated by $n = l = 0$. Otherwise, $1 \leq n \leq 1024$ and $1 \leq l \leq 1000$. Then follow n words of length l over the amino acid alphabet. They represent the leaves of a complete binary tree, from left to right.

The input must be read from standard input.

Output

For each test case, output a line containing some optimal common ancestor and the minimal total costs.

The output must be written to standard output.

Sample input	**Sample output**
4 3 AAG AAA GGA AGA 4 3 AAG AGA AAA GGA 4 3 AAG GGA AAA AGA 4 1 A R A R 2 1 W W 2 1 W Y 1 1 Q 0 0	AGA 3 AGA 4 AGA 4 R 2 W 0 Y 1 Q 0

D - Boolean Logic

Source file name: `boolean.c`, `boolean.cpp` *or* `boolean.java`

Propositions are logical formulas consisting of proposition symbols and connecting operators. They are recursively defined by the following rules:

1. All proposition symbols (in this problem, lower-case alphabetic characters, e.g., a and z) are propositions.
2. If P is a proposition, $(!P)$ is a proposition, and P is a direct subformula of it.
3. If P and Q are propositions, $(P\&Q)$, $(P|Q)$, $(P --> Q)$, and $(P <-> Q)$ are propositions, and P and Q are direct subformulas of them.
4. Nothing else is a proposition.

The operations !, &, |, $-->$, and $<->$ denote logical negation, conjunction, disjunction, implication, and equivalence, respectively. A proposition P is a subformula of a proposition R if $P = R$ or P is a direct subformula of a proposition Q and Q is a subformula of R.

Let P be a proposition and assign boolean values (i.e., 0 or 1) to all proposition symbols that occur in P. This induces a boolean value to all subformulas of P according to the standard semantics of the logical operators:

negation	conjunction	disjunction	implication	equivalence
$!0 = 1$	$0\&0 = 0$	$0 \mid 0 = 0$	$0 --> 0 = 1$	$0 <-> 0 = 1$
$!1 = 0$	$0\&1 = 0$	$0 \mid 1 = 1$	$0 --> 1 = 1$	$0 <-> 1 = 0$
	$1\&0 = 0$	$1 \mid 0 = 1$	$1 --> 0 = 0$	$1 <-> 0 = 0$
	$1\&1 = 1$	$1 \mid 1 = 1$	$1 --> 1 = 1$	$1 <-> 1 = 1$

This way, a value for P can be calculated. This value depends on the choice of the assignment of boolean values to the proposition symbols. If P contains n different proposition symbols, there are 2^n different assignments. To evaluate all possible assignments we may use truth tables.

A truth table contains one line per assignment (i.e., 2^n lines in total). Every line contains the values of all subformulas under the chosen assignment. The value of a subformula is aligned with the proposition symbol, if the subformula is a proposition symbol, and with the center of the operator otherwise.

Input

The input contains several test cases, each on a separate line. Every test case denotes a proposition and may contain arbitrary amounts of spaces in between. The input file terminates immediately after the newline symbol following the last test case.

The input must be read from standard input.

Output

For each test case generate a truth table for the denoted proposition. Start the truth table by repeating the input line. Evaluate the proposition (and its subformulas) for all assignments to its variables, and output one line for each assignment. The line must have the same length as the corresponding input line and must consist only of spaces and the characters 0 and 1. Output an empty line after each test case.

Let $s_1, \ldots, s_n$ be the proposition symbols in the denoted proposition sorted in alphabetic order. Then, all assignments of 0 to s_1 must precede the assignments of 1 to s_1. Within each of these blocks of assignments, all assignments of 0 to s_2 must precede the assignments of 1 to s_2, and so on.

The output must be written to standard output.

Sample input

```
((b --> a) <-> ((! a) --> (! b)))
  ((y &  a)   -  ->(c |c))
```

Sample output

```
((b --> a) <-> ((! a) --> (! b)))
  0  1  0   1    1 0   1   1 0
  1  0  0   1    1 0   0   0 1
  0  1  1   1    0 1   1   1 0
  1  1  1   1    0 1   1   0 1

  ((y &  a)   -  ->(c |c))
    0 0  0       1  0 00
    1 0  0       1  0 00
    0 0  0       1  1 11
    1 0  0       1  1 11
    0 0  1       1  0 00
    1 1  1       0  0 00
    0 0  1       1  1 11
    1 1  1       1  1 11
```

E - The Starflyer Agents

Source file name: `agents.c`, `agents.cpp` *or* `agents.java`

Famed investigator Paula Myo, working on behalf of the 2011 established Commonwealth government, is determined to stop the Starflyer from spying. The Starflyer is a ''human-friendly'' and powerful alien sentinel intelligence that was found by a space exploration frigate in the Dyson Alpha solar system in year 2285. It is not clear what the Starflyer's real intentions are towards the Commonwealth ... so, it is always better to be safe than sorry!!!

The Starflyer has the ability to control technological equipment; it typically infiltrates droids and uses them as agents. As a matter of fact, droids are carefully identified and tracked in the Commonwealth. Every droid has a history of software updates and each software update is tagged with a hash. A *hash* is a term built recursively from variable, constant, and function symbols as follows:

- any variable and any constant is a hash;
- if each $h_1, \ldots, h_k$ is a hash and f is a function symbol, then $f(h_1, \ldots, h_k)$ is a hash.

As a security measure, a well-kept secret from the general population, the Commonwealth enforces the following policy on droid software updates: for each droid, the tags of any software updates must be compatible. Two hashes h_1 and h_2 are *compatible* if there is a mapping θ from variables to hashes such that $h_1\theta = h_2\theta$, where $h_1\theta$ (resp., $h_2\theta$) denotes the simultaneous replacement of any occurrence of each variable x in h_1 (resp., h_2) with the hash $\theta(x)$. A sequence of hashes $h_1, \ldots, h_n$ is *compatible* if there is θ such that $h_1\theta, \ldots, h_n\theta$ are all equal.

For example, assume that X, Y, Z are variables, c, d are constants, and f, g are function symbols, and consider the hashes h_1, h_2, and h_3 as follows:

$$h_1 : f(X, g(c)) \qquad h_2 : f(f(Y), Z) \qquad h_3 : f(c, g(Y, d))$$

Observe that h_1 and h_2 are compatible because the mapping $\theta = \{X \mapsto f(Y), Z \mapsto g(c)\}$ satisfies $h_1\theta = h_2\theta$. However, any other pair from h_1, h_2, and h_3 is not compatible. Therefore, any sequence of hashes containing h_1, h_2, and h_3 is not compatible because there is no mapping θ such that $h_1\theta = h_2\theta = h_3\theta$.

Detective Myo has just been briefed on the aforementioned security policy. She strongly believes that the Starflyer infiltrates the droids via software updates without having any knowledge of the security policy. If her intuition is right, then this is the chance to detect and stop some Starflyer agents. You have been assigned to Myo's team: your task is to write an algorithm for determining if a sequence of hashes is compatible or not.

Can you help Detective Myo to uncover the Starflyer agents?

Input

The input consists of several test cases. The first line of each test case contains a string *name* and a natural number n separated by a blank ($2 \leq n \leq 20$, $1 \leq |name| \leq 16$). Then n lines follow, each containing a hash h_i ($1 \leq i \leq n$, $1 \leq |h_i| \leq 512$). You can suppose that:

- The *name* is an alphanumeric text (without blanks) that has a length less than or equal to 16 characters.
- Each one of the n hashes was built according to the above definition and has a length less than or equal to 512 characters.
- The variables, constants, and function symbols are formed exclusively from alphabetic characters. The first character of a variable symbol is an uppercase letter and the first character of a constant or function symbol is a lowercase letter.

The last test case is followed by a line with the text ''`END 0`''.

The input must be read from standard input.

Output

For each test case, a line must be printed. If the sequence of hashes $h_1, \ldots, h_n$ is compatible, then print the line

```
analysis inconclusive on XXX
```

or if the sequence of hashes $h_1, \ldots, h_n$ is not compatible, then print the line

```
XXX is a Starflyer agent
```

where `XXX` corresponds to *name* in the test case.

The output must be written to standard output.

Sample input	**Sample output**

`r2d2 3`
`f(X,g(c))`
`f(f(Y),Z)`
`f(c,g(Y,d))`
`c3po 2`
`f(X,g(c))`
`f(f(Y),Z)`
`PC2 2`
`f(f(Y),Z)`
`f(c,g(Y,d))`
`END 0`
	`r2d2 is a Starflyer agent`
	`analysis inconclusive on c3po`
	`PC2 is a Starflyer agent`

F - The Melding Plague

Source file name: `plague.c`, `plague.cpp` *or* `plague.java`

The *Nostalgia for Infinity* is an ancient ship that once carried hundreds of thousands, but now its crew is only a handful of Ultras --highly-modified humans adapted to the rigors of long interstellar spaceflight. And they are desperate to find a cure because most of their crew members are still infected with the *Melding Plague*, an alien virus that attacks human cells and machine nanotechnology in equal measure, perverting them into grotesque combinations. It is believed that some special mutations of the virus can help cure the dying Ultras.
The Ultras' patologists identified *protein configurations* as the essential constituents of the Melding Plague, some sort of genetic blueprint of the alien virus. Protein configurations are collections of proteins without any internal order and in which proteins can occur several times. For example, the following is the protein configuration last found in the infected blood of Nostalgia for Infinity's captain John Brannigan:

`POMC CAD CAD SCN5A XIRP2 SCN5A ELTD1.`

Protein configurations *mutate* according to the individual mutation of its proteins. In 1-step mutation *all* proteins in the configuration that can mutate indeed mutate, and those which cannot mutate stay the same. Mutations continue over and over, changing configurations step by step. Fortunately, Ultras' pathologists have identified protein configurations that are curable with appropriate therapies. Then, the hope for an Ultra infected with the Melding Plague is to have the protein configuration of the virus mutating to a curable protein configuration. Of course, therapies must be applied within a limit of mutation steps.
A *protein mutation* is described by an ordered pair of protein names (p, q) stating that protein p mutates to protein q. If $\mathcal{M} = \{(\texttt{CAD}, \texttt{CELR2}), (\texttt{ELTD1}, \texttt{XIRP2})\}$ is a collection of protein mutations, then the protein configuration of the virus in captain Brannigan's blood, depicted above, mutates by $\mathcal{M}$ in 1-step to the protein configuration:

`POMC CELR2 CELR2 SCN5A XIRP2 SCN5A XIRP2.`

Please remember that because the order in the protein configurations is immaterial, the first configuration can be written in 1260 different ways, and the 1-step mutation just shown in 630 different ways.
Your task today is to help the surviving Ultras by building a program that, given a collection of protein mutations $\mathcal{M}$, an initial protein configuration I, a cure protein configuration C, and a natural number n representing a search bound:

- if I mutates to C within at most n steps, computes the minimal number of such mutation steps;
- otherwise, it must inform the Ultras that I cannot mutate to C within n steps.

To easy your burden, Ultras' pathologist are providing you with extra knowledge: they have identified that if a cure by this method exists, one need to consider only *deterministic* mutations $\mathcal{M}$, i.e., if (p, q_1) and (p, q_2) are in $\mathcal{M}$, then $q_1 = q_2$.

Input

The input consists of several test cases. A test case begins with a line containing four natural numbers $N_{\mathcal{M}}$, N_I, N_C, and n separated by a blank, and with $0 \leq N_{\mathcal{M}}, N_I, N_C, n \leq 1000$.
If $N_{\mathcal{M}}$, N_I, and N_C are greater than 0, then $N_{\mathcal{M}} + N_I + N_C$ lines follow. The first $N_{\mathcal{M}}$ lines define the collection $\mathcal{M}$ of *protein mutations* in which each line consists of a pair of strings p and q separated by a blank, representing protein mutation (p, q). Each one of the following N_I lines consist of a string p and a natural number i separated by a blank, representing the number of occurrences i of protein p in the *initial protein configuration* I. Each one of the last N_C lines consist of a string q and a natural number c separated by a blank, representing the number of occurrences c of protein q in the *cure protein configuration* C. The natural number n defines the *search bound*.
The input ends with $N_{\mathcal{M}} = N_I = N_C = n = 0$.

The input must be read from standard input.

Output

For each test case your program must output exactly one line as follows:

- if $\mathcal{M}$ is not deterministic, then output:

  `Protein mutations are not deterministic`

- if $\mathcal{M}$ is deterministic and I mutates to C by $\mathcal{M}$ in at most n mutation steps with a minimum of k mutation steps, then output:

  `Cure found in` k `mutation(s)`

- otherwise output:

  `Nostalgia for Infinity is doomed`

The output must be written to standard output.

Sample input	**Output for the sample input**
2 5 4 3 CAD CELR2 ELTD1 XIRP2 POMC 1 CAD 2 SCN5A 2 XIRP2 1 ELTD1 1 POMC 1 CELR2 2 SCN5A 2 XIRP2 2 2 3 3 3 GP183 NALCN CAC1S GP183 CAC1S 2 YCFI 1 MRP6 3 YCFI 1 MRP6 3 NALCN 2 2 3 3 1 GP183 NALCN CAC1S GP183 CAC1S 2 YCFI 1 MRP6 3 YCFI 1 MRP6 3 NALCN 2 3 2 1 2 CAD YCFI ELTD1 XIRP2 CAD SCN5A CAD 1 YCFI 1 YCFI 2 0 0 0 0	Cure found in 1 mutation(s) Cure found in 2 mutation(s) Nostalgia for Infinity is doomed Protein mutations are not deterministic

G - Code Theft

Source file name: `code.c`, `code.cpp` *or* `code.java`

While reviewing code recently being checked into the repository, Jim discovered that some employees now and then seemed to copy code right from the Internet into the company code base. This would be a potential disaster as the company then risks getting sued by copyright holders of the original code. The obvious solution, talking to the employees and kindly ask them not to submit any stolen code, seemed to solve the problem. Still, it was decided that a screening process should be introduced to detect newly stolen code.

The screening would work as follows: Every time new code was checked in, the full contents of the changed les where matched against a repository of known open source code. For each le the longest match, in number of consecutive lines, should be reported.

Comparison is done line by line. Empty lines, and lines only containing space, are ignored during comparison and not counted. Leading and trailing spaces should be ignored completely and consecutive space characters inside the lines are treated as one single space. The comparison is case-sensitive.

Input

Test data starts with the number $0 \leq N \leq 100$ of code fragments in the repository. Then follows, for each code fragment, one line containing the file name that the fragment was fetched from and the contents of the fragment on subsequent lines. File names will neither contain whitespace nor be guaranteed to be unique. The name is at most 254 characters long. Each fragment is terminated by `***END***` on a line by its own. This line is not considered being part of the fragment.

After the fragments in the repository have all been listed, comes the actual code snippet to nd matches for. This snippet is also terminated by `***END***` on a line by its own.

Lines are guaranteed to be no longer than 254 characters. No code fragment will be longer than 10000 lines. Any code and file name lines will only contain the ASCII characters 32-126. The total size of the input le will not exceed 10^6 characters.

The input must be read from standard input.

Output

For each test case, write the number of matching consecutive lines (empty lines not counted) in a longest match from the repository, followed by a space-separated list of the le names of each fragments containing a match of this length, given in the order that the matching fragments were presented in the repository description. If no fragments match, write the number 0 on a line of its own.

The output must be written to standard output.

Sample input	**Sample output**

2
HelloWorld.c
int Main() {
printf(‘‘Hello %d\n’’,i);
}
END
Add.c
int Main() {
for (int i=0; i<10; i++)
sum += i;
printf(‘‘SUM %d’’, sum);
}
END
int Main() {
printf(‘‘Hello %d\n’’,i);
printf(‘‘THE END\n’’);
}
END
	2 HelloWorld.c

H - Anagram Groups

Source file name: `anagram.c`, `anagram.cpp` *or* `anagram.java`

World-renowned Prof. A. N. Agram's current research deals with large anagram groups. He has just found a new application for his theory on the distribution of characters in English language texts. Given such a text, you are to find the largest anagram groups.
A text is a sequence of words. A word w is an anagram of a word v if and only if there is some permutation p of character positions that takes w to v. Then, w and v are in the same anagram group. The size of an anagram group is the number of words in that group. Find the 5 largest anagram groups.

Input

The input contains words composed of lowercase alphabetic characters, separated by whitespace. It is terminated by EOF.

The input must be read from standard input.

Output

Output the 5 largest anagram groups. If there are less than 5 groups, output them all. Sort the groups by decreasing size. Break ties lexicographically by the lexicographical smallest element. For each group output, print its size and its member words. Sort the member words lexicographically and print equal words only once.

The output must be written to standard output.

Sample input	**Sample output**
undisplayed trace tea singleton eta eat displayed crate cater carte caret beta beat bate ate abet	Group of size 5: caret carte cater crate trace . Group of size 4: abet bate beat beta . Group of size 4: ate eat eta tea . Group of size 1: displayed . Group of size 1: singleton .