# FAU-Programmierwettbewerb 2005

25. Juni 2005

Problemübersicht:



Viel Spaß beim Programmieren!

Die Dokumentation des Java-API ist unter
`http://www2.informatik.uni-erlangen.de/Dokumentation/j2sdk-1.4.2/api/`
einzusehen.

Die Dokumentation der Standard Template Library ist unter
`http://www2.informatik.uni-erlangen.de/Dokumentation/STL/` zu finden.

Verwendete Compiler und ihre Optionen:
C: `gcc -Wall -lm`, Version: 3.3.3
C++: `g++ -Wall`, Version: 3.3.3
Java: ????

# Problem A
# Happy Countries

Author: Dominik Scheder

**Time Limit – 2 seconds**

Back in his first year of highschool, young Dominik was very interested in Geography. In his first lesson, he learned that the Soviet Union - yes, it still existed back then - was by far the largest country on earth. But, as a matter of fact, its population is much smaller than China's, which is the country with the biggest population. "The Soviet Union can't be a happy country", young Dominik thought. "All those huge plains with no one living there...", but he was also telling himself "China can't be happy either! So many people but far too little space for them". So he thought in order to be happy, a country should have the same rank if measured by population as if measured by area. For example, the United States is a happy country: it's third by both area and population! Your task is, given a list of countries that shows their area and their population, to determine all happy countries.

### Input

The input consists of several testcases. Each testcase begins with a line containing a single integer, $1 \leq n \leq 1000$, the number of countries in this testcase. This is followed by $n$ pairs of lines, the first of each containing the name of a country, the second the area. This list is followed by a blank line and by a list describing, in exactly the same format as above, the population of the countries. Each testcase is followed by a blank line. The input is ended by a testcase where $n = 0$, which should not be processed. You can assume that the set of names in the first list and the set of names in the second list are identical. Also, all numbers in the input will be within the range of 32-bit int, and there are no two different countries with the same size of area or the same number of people.

### Output

For each testcase, output the names of the happy countries, in descending order, one name per line. If there is no happy country, write "There is no happy country!". After each testcase, output a blank line.

### Sample Input

```
3
United States of America
100
People's Republic of China
95
Brazil
80

People's Republic of China
1000
Brazil
250
United States of America
300
```

### Sample Output

```
Brazil
```

# Problem B
# Just Link It

Author: Christian Riess

**Time Limit – 5 seconds**

The usual process of creating binary code involves compiling and linking. Compiling only means to translate code from a source language to a destination language. In our case this means that machine executable code is created out of our programming language. The second step in creating a machine executable program is to link this executable code to external libraries. Since linking is a separate work step, there often exists a linker in addition to the compiler. For us - as we are old fashioned Linux hackers - the linker of our choice is the GNU linker, invoked by the command "ld".

The pesky part of that linker are the little details: If we want to link our program P against a library A and a library B, but library B also depends on library A, we have to take care in which order we list the libraries A and B.

Example: "**ln -lA -lB P**" would link our program P correctly, because B depends on A and A is already included when B is treated. But "**ln -lB -lA P**" would fail, because at the time B is processed, it still contains unknown symbols from A, which cannot be resolved. Circular dependencies can be resolved by repeating the entries:

"**ln -lB -lA -lB P**" would also work in our previous example, and so could these be resolved. Since it is quite annoying to work this out with more than just two libraries, it is your job - given the library dependencies - to print the correct linker command line for that problem. Since circular dependencies are more complicated, we simply want an error message, if you detect a circular dependency. Print those include libraries at first which depend on no other libraries in alphabetical order, afterwards those which depend on nothing but the already printed include libraries (again in alphabetical order), and so on.

### Input

The input consists of the number of test cases $c, 0 < c \leq 1000$. Each test case starts with a line containing the program name, the number of involved libraries $l$, $0 \leq l \leq 26$, and the number of dependencies $d, 0 \leq d \leq 1000$. You can assume that the names of the libraries correspond to the uppercase letters of the Latin alphabet, starting with "A", so the existence of three libraries means there are the libraries "A", "B" and "C". This line is followed by d lines describing one dependency, given by two letters separated by a space, where the first one denotes the library that is dependent on the second. You can assume that the program name will not be longer than 20 characters. There won't be a reflexive dependency of the kind "A" depends on "A", and there are only dependencies between libraries given that occur in the current linking process, i.e. if there are only two libraries, no dependency with a library "C" will occur. The lines describing the dependencies are not necessarily unique, but there are never more than 1000 lines of dependencies per test case.

### Output

For each test case your program should print the linker command line for the given problem, in the order as described above. Every linker command line should be written in a separate line. If there is a circular dependency, output a single line containing "error: circular dependency!"

### Sample Input

```
3
icpc_solution.o 1 0
hal_core.o 3 2
A B
C B
everyday_code.o 2 2
A B
B A
```

### Sample Output

```
ln -lA icpc_solution.o
ln -lB -lA -lC hal_core.o
error: circular dependency!
```

# Problem C
# A graph problem

Author: Der General

**Time Limit – 2 seconds**

Given an undirected graph of the following form with $n$ nodes, $1 \leq n \leq 76$:


Your task is to calculate the number of subsets of nodes of the graph with the following properties:

- no nodes in the subset should be connected
- it shouldn't be possible to add further nodes to the subset without violating the first condition

For a graph with 5 nodes the number of subsets which fulfill the above conditions is 4. The subsets are $\{1, 3, 5\}, \{2, 4\}, \{2, 5\}, \{1, 4\}$.

### Input

The input will consist of a sequence of numbers $n$, $1 \leq n \leq 76$. Each number will be on a separate line. The input will be terminated by EOF.

### Output

Output the number of subsets as described above on a single line. The number of all subsets will be less than $2^{31}$.

### Sample Input

```
1
2
3
4
5
30
```

### Sample Output

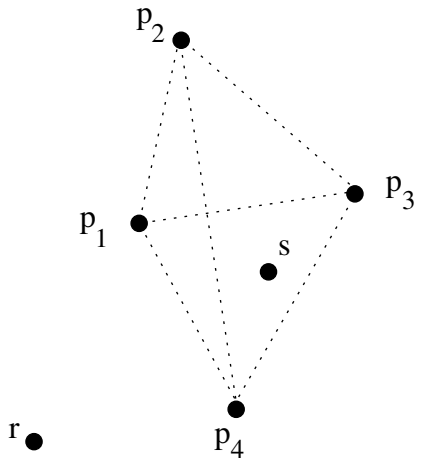
```
1
2
2
3
4
4410
```

# Problem D
# Points

Author: Tilmann Spiegelhauer

**Time Limit – 10 seconds**

In this problem you will be given a set of points in the Euclidian plane. The number of points in the set will never exceed 100000. The coordinates of these points will be integer coordinates and will have an absolute value smaller than or equal to 10000. There will be no identical points in the first set. Then you will be given a second set of points. The points in the second have integer coordinates, too. For each point in the second set you will have to determine whether it is contained in a triangle spanned by three points in the first set. A point on the edge of a triangle is considered to be "inside" the triangle. In the following example the points $p_1, p_2, p_3, p_4$ belong to the first set. The points $r$ and $s$ belong to the second set. The point $r$ isn't contained in any triangle spanned by three points of the first set. The point $s$ is contained in two triangles. For example, the triangle spanned by $p_2, p_3, p_4$.




## Input

You will be given several testcases. A testcases consists of the number of points $p$, $3 \leq p \leq 100000$ in the first set. It is followed by $p$ pairs of numbers, each describing a point of the first set, the first number of a pair denoting the $x$-coordinate of the point, the second the $y$-coordinate. Each pair is on a seperate line. There may be colinear points in the first set. The next number in the input gives you the number of points $r$ in the second set. It is followed by $r$ pairs of numbers, each describing a point, each on a separate line. The first number of a pair being the $x$-coordinate, the second number being the $y$-coordinate of the point. All coordinates in the input will be integer coordinates.

## Output

For each point in the second set, output if the point lies in a triangle spanned by three points of the first set. If the point lies inside a triangle output `inside` otherwise output `outside`.

### Sample Input

```
4
0 0
4 4
0 4
4 0
6
2 2
4 4
1 1
0 2
0 10
10 0
```

### Sample Output

```
inside
inside
inside
inside
outside
outside
```

# Problem E
# Polynomial Marriage

Author: Tilmann Spiegelhauer

**Time Limit – 2 seconds**

You might know the mathematical creatures which go by the name polynomials. In this case you will have to deal with a certain tribe of them, the polynomials of the tribe $\mathbb{Z}[X]$. Two polynomials of this tribe can only marry, if the male polynomial divides the female polynomial without leaving a remainder. Given a male polynomial and a female polynomial your task is to decide if they can marry. The leading coefficient of the male polynomial will always be 1 or -1, so the result of the division will always be in $\mathbb{Z}[X]$.

### Input

There will be an even number of lines in the input. There will be a polynomial in each line. The polynomials on the first, third line and on so on and so forth will be female polynomials. The other polynomials will be male polynomials. The description of a polynomial will consist of the degree $d, 1 \leq d \leq 10$, followed by the $d+1$ coefficients in decreasing order, the first coefficient will be the leading coefficient. The coefficients of the result will always fit into an int. A coefficients' absolute value will be at most 10. A polynomial will have at least degree one. The input will be terminated by EOF.

### Output

Output the remainder of the division for each pair of polynomials on a line in the format specified in the Sample Output. Output a 0 for the zero polynomial. In all other cases omit monomials with the coefficient zero. Do not print the coefficient 1 for monomials of degree 1 or greater.

### Sample Input

```
1 2 0
1 -1 0
3 1 1 0 -1
2 1 0 0
3 1 1 0 2
2 1 0 0
3 1 1 -1 0
2 -1 0 0
3 1 1 1 0
2 -1 0 0
3 1 1 0 1
3 1 0 0 0
10 -2 -1 -3 0 -1 -3 -2 -1 -1 1 -3
5 1 -3 0 2 -3 -1
```

### Sample Output

```
0
-1
2
-x^1
x^1
x^2 + 1
-1652x^4 + 165x^3 + 478x^2 - 1903x^1 - 572
```

# Problem F
# Protein similarity

Author: Thorsten Meinl

**Time Limit – 10 seconds**

Proteins are responsible for almost everything that goes on in our body. All proteins are built up of only twenty amino acids which are chained together in countless variations. It is common to abbreviate the amino acids by a three letter code, e.g. `SER` for Serin, `LEU` for Leucin, etc. One of the interesting tasks in bioinformatics is to determine the similarity of two proteins. One way of doing so, is to align the proteins and compute a score for the alignment. Take e.g. the two amino acid sequences `SER SER CYS ASP LEU SER CYS` and `SER CYS CYS ASP ASP LEU CYS SER ASP ASP CYS`. One possible alignment is

```
SER CYS CYS ASP ASP LEU CYS SER ASP ASP CYS
 |       |       |   |       |           |
SER SER CYS ___ ASP LEU ___ SER ___ ___ CYS
```

As you can see, there are amino acids that **match** in both sequences (e.g. `SER` at the beginning), amino acids that **do not match** (`CYS` and `SER` at the second position) and **gaps** in either of the sequences (indicated by `___`). Each of the three situations gets a distinct score value. Suppose a match gets a score of 2, a mismatch a score of -1 and a gap a score of -2 (also called *gap penalty*) then the overall score of the alignment above is 3. The higher the score the more similar the two sequences are. Of course there is more than just one alignment for both sequences resulting in different scores. Your task will be to find out the score for the best alignment. Please note that in the final alignment both the shorter and the longer sequence may be filled up with gaps.

### Input

The input consists of a bunch of test cases. Each test case is built up of three lines. In the first line the three scores are given as integer values separated by a space (the match score is always non-negative, the other two are always non-positive). The first value is the match score, the second the mismatch score and the third the gap penalty. Then the two proteins follow in the next two lines. They are given as the chain of the three letter codes of their amino acids. There are a maximum of twenty different amino acids. The last test case is terminated by EOF.

The sequences will not be longer than 10.000 amino acids.

### Output

Your program should output the maximal score for the alignment of each sequence pair as an integer value, each in its own line.

### Sample Input

```
1 0 0
SERCYSCYSASPASPLEUCYSSERASPASPCYS
SERSERCYSASPLEUSERCYS
2 -1 -2
SERCYSCYSASPASPLEUCYSSERASPASPCYS
SERSERCYSASPLEUSERCYS
```

### Sample Output

```
6
3
```

# Problem G
# Slow Sort

Author: Christian Riess

**Time Limit – 2 seconds**

Part of the program of the university's foundation course is to study sorting algorithms. We all know heapsort as a powerful "in situ" algorithm, and we know that quicksort is the striking solution for average case input. We also appreciate bubblesort for its pleasant analytical structure, even if it is not very efficient compared to the ones previously mentioned.

An even slower method of sorting than bubblesort is slowsort. Slowsort is very, very slow even compared to the rather slow bubblesort. It works the following way:

```
while (not_sorted(current_permut))
        current_permut = get_next_permutation()
```

where `get_next_permutation()` simply creates one permutation after another from the input string, according to following rule:

If the initial state is `1 2 3 4` then the consecutive return values of `get_next_permutation()` are

```
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

Since for one of these permutations of the input should be correctly sorted (this means all elements are in increasing order), the test `not_sorted(current_permut)` fails sometime and we know which permutation to apply for sorting the input. Now before we are running some tests on our algorithm, we are interested in the number of loops the algorithm will take. It is your job to calculate this number.

### Input

The input consists of the number of test cases $c, 1 \leq c \leq 100$. Each test case consists of one line containing a number $n, 1 \leq n \leq 12$, followed by n numbers which form a permutation of the numbers $k, 1 \leq k \leq n$. All numbers in one line are divided by a single space. The given permutation represents the state in which the input is sorted, and it is yours to calculate the number of steps there from the initial configuration, which consist simply of $n$ increasing numbers. In a situation as given in the example above, the permutation `1 3 2 4` would cause the loop to execute twice, since it is the second permutation after the initial state.

### Output

For each test case print one line containing the number of iterations until the input is sorted. You can assume that this number will always fit into an integer.

### Sample Input

```
2
4 1 4 2 3
3 1 3 2
```

### Sample Output

```
4
1
```

> "Computers these days are trying to bury us with useless information." I grumbled. "I think that they have entirely too much time on their hands and are using it to make us miserable. I'm trying to think of some way to keep them busy so that they will leave us humans alone." (David Condic)

For references, please visit:

`http://portal.acm.org/citation.cfm?id=101139.101144` "Junk facts and the Slowsort"
`http://www.stefan-baur.de/cs.algo.slowsort.html` "SlowSort implementation in Prolog"

# Problem H

## The snake game

Author: Ingrid Fischer

**Time Limit – 2 seconds**

There is a new toddler game on the market, that parents are buying like crazy. It is a cute little card game. You start with a designated card and replace one card by two other cards until you have a looooooooooooooong snake your toddler likes. To have more fun you can make up the rules how cards can be replaced by yourself, the only restriction is, that one card has to be substituted by two cards. Otherwise your toddler is disappointed because the snake is not growing. Assume the starting card is called *Snake* and you have the following snake replacement rules:

```
Snake replaced with Head Tail
Head  replaced with Head Body
Body  replaced with Body Body
```

Then the following snakes can be built:

```
Head Body Body Body Body Tail
Head Body Body Body Tail
Head Body Body Tail
Head Body Tail
Head Tail
```

Of course even longer snakes are possible. Your task is now to decide whether a set of given snakes can be built from a set of snake rules. To make it for grown-ups like you more interesting, you should give the number of possibilities to construct a snake from the rules.

### Input

The input consists of two parts. First the rules are given and second snakes are given. The rules part starts with an integer in the first line saying how many rules are following. Each rule is written in a separate line. A rule starts with the name of the card to be replaced followed by the names of the two cards it can be replaced with. One card can be only replaced by two other cards. In the line following the rule definitions an integer is given with the number of snakes possibly built from the rules. Each snake is written in one line.

### Output

For each possible snake given in the input it must be decided whether this snake can be built with the help of the given rules. If a snake can be accepted the output is the number of ways to construct the snake, otherwise it is ¡i¿No¡/i¿. Each decision is printed in a new line.

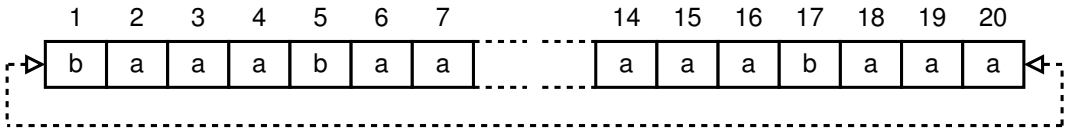
| Sample Input | Sample Output |
|---|---|
| 3<br>Snake Head Tail<br>Head Head Body<br>Body Body Body<br>6<br>Head Body Wing Tail<br>Head Body Body Body<br>Head Body Body Tail<br>Body Head Body Tail<br>Head Body Tail<br>Head Tail | No<br>No<br>2<br>No<br>1<br>1 |

# Problem I
# The Thuringian Machine

Author: Stefan Büttcher

**Time Limit – 10 seconds**

The Thuringian Machine is very similar to the well-known Turing Machine. One of the major differences is that – due to the lack of military funding in the post-cold-war world – the Thuringian Machine does not have an infinite tape. Instead, its tape consists of 20 cells that are arranged in a circular fashion. So, whenever the read/write head is above cell 20 and receives the command to move to the right, it goes to cell 1. Analogously, when it is above cell 1 and receives the command to move to the left, its new position is above cell 20. While this limitation makes the Thuringian Machine computationally incomplete :(, it allows you to solve the Halting Problem :)




At every point in time, the *configuration* of the Thuringian Machine is defined by the contents of its tape $T_1 \dots T_{20} \in \{a, b\}^{20}$, the current head position $h$ (index of the cell above which the read/write head is), and an integer number $s$ - the state of the Thuringian Machine.

A program for the Thuringian Machine is a finite number of lines that define a partial function mapping from *(machine state, cell content)* pairs to *(machine state, cell content, head movement)* triples. For instance, the line

```
1 a 2 b right
```

says that when the TM is in state 1 and reads an "a" from the current tape cell, it writes an "b" to the current cell, switches to state 2 and moves the tape head one cell to the right. Similarly, the line

```
7 b 7 b left
```

tells the TM to leave the cell content untouched, when it is in state 7 and reads a "b", and to move the tape head one cell to the left. Finally,

```
3 b 1 a stay
```

says that when the Thuringian Machine is in state 3 and reads a "b", it writes an "a", switches to state 1, and does not move the tape head.

At the beginning, the TM is always in state 1, and the tape head is above cell 1. State 0 is a special state: When the Thuringian Machine enters state 0 (either explictly or implicitly), it stops the program execution. Whenever the machine encounters a configuration for which there is no valid rule, it enters state 0 and terminates (this is the implicit case).

Your task is to write a program that, given the description of a Thuringian Machine and the tape content, determines whether the machine terminates on the given tape. In other words, you have to solve the Thuringian Halting Problem.

### Input

The first input line contains an integer $n$, the number of test cases to follow. Each test case consists of the description of a TM program and the initial tape content, as shown in the Sample Input below. It starts with two integers $s$ and $r$ (the number of different states (not including the halting state) and the number of rules (transitions), respectively). The following $r$ lines each contain a single transition rule. The $(r + 1)$-th line contains the initial content of the TM's tape. It is guaranteed that each Thuringian Machine is deterministic (i.e., at most one transition for every *(machine state, cell content)* pair) and has no more than 40 different states.

### Output

For each test case, print one line, containing either the string "Terminates." or "Does not terminate.", depending on whether the TM halts on the given input string.

### Sample Input

```
3
2 3
1 a 2 a right
1 b 2 a right
2 b 2 b right
bbbbbbbbbbbbbbbbbbbb
1 2
1 a 1 a right
1 b 1 b right
abababababababababab
5 8
1 a 2 a right
1 b 2 b right
2 a 2 a right
2 b 3 b left
3 a 4 b left
4 a 4 a left
4 b 5 b right
5 a 2 b right
baaaaaaaaaaaaaaaaaab
```

### Sample Output

```
Terminates.
Does not terminate.
Terminates.
```