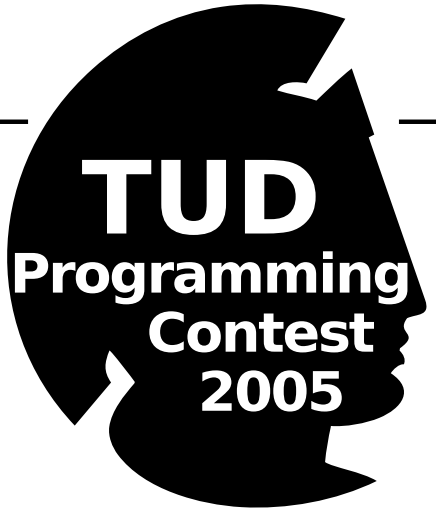



http://tud-pc.informatik.tu-darmstadt.de

Saturday, June 25th 2005

# PROBLEM SET

FOR MAIN CONTEST

**This problem set should contain 9 (nine) problems on 14 (fourteen) numbered pages. Please inform a runner immediately if something is missing from your problem set.**

# Contents



# General Remark

- All programs read their input from `stdin` and output to `stdout` (no files allowed). `stderr` can be used for test outputs. You can rely on correct input specifications (no error handling code on input required).
- In Java solutions, the `main` method has to be in a public class named `Solution`.
- If not specified otherwise, all numbers can be assumed to fit into a 32-bit signed integer.

Have fun!

# 1 Stamps

## Background

Everybody hates Raymond. He's the largest stamp collector on planet earth and because of that he always makes fun of all the others at the stamp collector parties. Fortunately everybody loves Lucy, and she has a plan. She secretly asks her friends whether they could lend her some stamps, so that she can embarrass Raymond by showing an even larger collection than his.

## Problem

Raymond is so sure about his superiority that he always tells how many stamps he'll show. And since Lucy knows how many she owns, she knows how many more she needs. She also knows how many friends would lend her some stamps and how many each would lend. But she'd like to borrow from as few friends as possible and if she needs too many then she'd rather not do it at all. Can you tell her the minimum number of friends she needs to borrow from?

## Input

The first line contains the number of scenarios. Each scenario describes one collectors party and its first line tells you how many stamps (from 1 to 1000000) Lucy needs to borrow and how many friends (from 1 to 1000) offer her some stamps. In a second line you'll get the number of stamps (from 1 to 10000) each of her friends is offering.

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print a single line with the minimum number of friends Lucy needs to borrow stamps from. If it's impossible even if she borrows everything from everybody, write `impossible`. Terminate the output for the scenario with a blank line.

## Sample Input

```
3
100 6
13 17 42 9 23 57
99 6
13 17 42 9 23 57
1000 3
314 159 265
```

## Sample Output

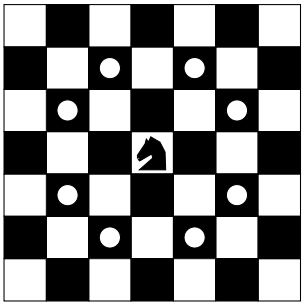
```
Scenario #1:
3

Scenario #2:
2

Scenario #3:
impossible
```

# 2 A Knight's Journey

## Background

The knight is getting bored of seeing the same black and white squares again and again and has decided to make a journey around the world. Whenever a knight moves, it is two squares in one direction and one square perpendicular to this.

The world of a knight is the chessboard he is living on. Our knight lives on a chessboard that has a smaller area than a regular $8 \times 8$ board, but it is still rectangular. Can you help this adventurous knight to make travel plans?



*The eight possible moves of a knight*

## Problem

Find a path such that the knight visits every square once. The knight can start and end on any square of the board.

## Input

The input begins with a positive integer $n$ in the first line. The following lines contain $n$ test cases.

Each test case consists of a single line with two positive integers $p$ and $q$, such that $1 \leq p \cdot q \leq 26$. This represents a $p \times q$ chessboard, where $p$ describes how many different square numbers $1, \ldots, p$ exist, $q$ describes how many different square letters exist. These are the first $q$ letters of the Latin alphabet: $\mathrm{A}, \ldots$

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print a single line containing the *lexicographically first* path that visits all squares of the chessboard with knight moves followed by an empty line. The path should be given on a single line by concatenating the names of the visited squares. Each *square name* consists of a *capital letter* followed by a *number*.

If no such path exist, you should output `impossible` on a single line.

## Sample Input

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

## Sample Output

```
Scenario #1:
A1

Scenario #2:
impossible

Scenario #3:
A1B3C1A2B4C2A3B1C3A4B2C4
```

# 3 Line Segments

## Background

Line segments are a very common element in computational geometry. A line segment is the set of points forming the shortest path between two points (including those points). Although they are a very basic concept it can be hard to work with them if they appear in huge numbers unless you have an efficient algorithm.

## Problem

Given a set of line segments, count how many distinct pairs of line segments are overlapping. Two line segments are said to be *overlapping* if they intersect in an infinite number of points.

## Input

The first line contains the number of scenarios.

Each scenario starts with the number $n$ of line segments ($1 \leq n \leq 100000$). Then follow $n$ lines consisting of four integers $x_1, y_1, x_2, y_2$ in the range $[0, 1000000]$ each, representing a line segment that connects the points $(x_1, y_1)$ and $(x_2, y_2)$. It is guaranteed that a line segment does not degenerate to a single point.

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print a single line containing the number of distinct pairs of overlapping line segments followed by an empty line. **Hint:** The number of overlapping pairs may not fit into a 32-bit integer.

## Sample Input

```
2
8
1 1 2 2
2 2 3 3
1 3 3 1
10 0 20 0
20 0 30 0
15 0 25 0
50 0 100 0
70 0 80 0
1
0 0 1 1
```

## Sample Output

```
Scenario #1:
3

Scenario #2:
0
```

# 4 Pimp My Ride

## Background

Today, there are quite a few cars, motorcycles, trucks and other vehicles out there on the streets that would seriously need some refurbishment. You have taken on this job, ripping off a few dollars from a major TV station along the way.

Of course, there's a lot of work to do, and you have decided that it's getting too much. Therefore you want to have the various jobs like painting, interior decoration and so on done by garages. Unfortunately, those garages are very specialized, so you need different garages for different jobs. More so, they tend to charge you the more the better the overall appearance of the car is. That is, a painter might charge more for a car whose interior is all leather. As those "surcharges" depend on what job is done and which jobs have been done before, you are currently trying to save money by finding an optimal order for those jobs.

## Problem

Individual jobs are numbered 1 through $n$. Given the base price $p$ for each job and a surcharge $s$ (in US$) for every pair of jobs $(i, j)$ with $i \neq j$, meaning that you have to pay additional $\$s$ for job $i$, if and only if job $j$ was completed before, you are to compute the minimum total costs needed to finish all jobs.

## Input

The first line contains the number of scenarios. For each scenario, an integer number of jobs $n$, $1 \leq n \leq 14$, is given. Then follow $n$ lines, each containing exactly $n$ integers. The $i$-th line contains the surcharges that have to be paid in garage number $i$ for the $i$-th job and the base price for job $i$. More precisely, on the $i$-th line, the $i$-th integer is the base price for job $i$ and the $j$-th integer ($j \neq i$) is the surcharge for job $i$ that applies if job $j$ has been done before. The prices will be non-negative integers smaller than or equal to 100000.

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print a single line:

"`You have officially been pimped for only $p`"

with `p` being the minimum total price. Terminate the output for the scenario with a blank line.

## Sample Input

```
2
2
10 10
9000 10
3
14 23 0
0 14 0
1000 9500 14
```

## Sample Output

```
Scenario #1:
You have officially been pimped for only $30

Scenario #2:
You have officially been pimped for only $42
```

# 5 Scavenger Hunt

## Background

Bill has been the greatest boy scout in America and has become quite a superstar because he always organized the most wonderful scavenger hunts (you know, where the kids have to find a certain route following certain hints). Bill has retired now, but a nationwide election quickly found a successor for him, a guy called George. He does a poor job, though, and wants to learn from Bill's routes. Unfortunately Bill has left only a few notes for his successor.

## Problem

Bill never wrote down his routes completely, he only left lots of little sheets on which he had written two consecutive steps of the routes. He then mixed these sheets and memorized his routes similarly to how some people learn for exams: practicing again and again, always reading the first step and trying to remember the following. This made much sense, since one step always required something from the previous step.

George however would like to have a route written down as one long sequence of all the steps in the correct order. Please help him make the nation happy again by reconstructing the routes.

## Input

The first line contains the number of scenarios. Each scenario describes one route and its first line tells you how many steps ($3 \leq S \leq 333$) the route has. The next $S - 1$ lines each contain one consecutive pair of the steps on the route separated by a single space. The name of each step is always a single string of letters.

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print $S$ lines containing the steps of the route in correct order. Terminate the output for the scenario with a blank line.

## Sample Input

```
2
4
SwimmingPool OldTree
BirdsNest Garage
Garage SwimmingPool
3
Toilet Hospital
VideoGame Toilet
```

## Sample Output

```
Scenario #1:
BirdsNest
Garage
SwimmingPool
OldTree

Scenario #2:
VideoGame
Toilet
Hospital
```

# 6 A Bug's Life

## Background

Professor Hopper is researching the sexual behavior of a rare species of bugs. He assumes that they feature two different genders and that they only interact with bugs of the opposite gender. In his experiment, individual bugs and their interactions were easy to identify, because numbers were printed on their backs.

## Problem

Given a list of bug interactions, decide whether the experiment supports his assumption of two genders with no homosexual bugs or if it contains some bug interactions that falsify it.

## Input

The first line of the input contains the number of scenarios. Each scenario starts with one line giving the number of bugs (at least one, and up to 2000) and the number of interactions (up to 1000000) separated by a single space. In the following lines, each interaction is given in the form of two distinct bug numbers separated by a single space. Bugs are numbered consecutively starting from one.

## Output

The output for every scenario is a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1, followed by one line saying either "`No suspicious bugs found!`" if the experiment is consistent with his assumption about the bugs' sexual behavior, or "`Suspicious bugs found!`" if Professor Hopper's assumption is definitely wrong.

## Sample Input

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

## Sample Output

```
Scenario #1:
Suspicious bugs found!

Scenario #2:
No suspicious bugs found!
```

# 7 Rdeaalbe

## Background

As you probably know, the human information processor is a wonderful text recognizer that can handle even sentences that are garbled like the following:

```
The ACM Itrenntaoial Clloegaite Porgarmmnig Cnotset (IPCC)
porvdies clolgee stuetnds wtih ooppriuntetiis to itnrecat
wtih sutednts form ohetr uinevsrtieis.
```

## Problem

People have claimed that understanding these sentences works in general when using the following rule: The first and last letters of each word remain unmodified and all the characters in the middle can be reordered freely.

Since you are a skeptical ACM programmer, you immediately set on to write the following program: Given a sentence and a dictionary of words, how many different sentences can you find that could potentially be mapped to the same encoding?

## Input

The first line contains the number of scenarios. Each scenario begins with a line containing the number $n$ of words in the dictionary ($0 \leq n \leq 10000$), which are printed on the following $n$ lines. After this there is a line containing the number $m$ of sentences that should be tested with the preceding dictionary ($0 \leq m \leq 10000$) and then $m$ lines containing those sentences. The sentences consist of letters from `a..z`, `A..Z` and spaces only and have a maximal length of 10000 characters. For each word in the dictionary a limitation of 100 characters can be assumed.

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. For each sentence output the number of sentences that can be formed on an individual line. It is guaranteed that this number can be expressed using a signed 32-bit datatype. Terminate the output for the scenario with a blank line.

## Sample Input

```
2
3
ababa
aabba
abcaa
2
ababa
abbaa
14
bakers
brakes
breaks
binary
brainy
baggers
beggars
and
```

```
in
the
blowed
bowled
barn
bran
1
brainy bakers and beggars bowled in the barn
```

## Sample Output

```
Scenario #1:
2
2

Scenario #2:
48
```

# 8 Acid Text

## Background

A couple of months ago the web standards project (WaSP) has come up with a test for modern browsers and their CSS implementation called acid2. This test ensures that all the browsers have similar results when it comes to parsing and displaying cascaded style sheet files (CSS) for HTML. Since you want to beat all the other text-based browsers on standard compliance you directly start implementing the CSS capabilities into your favorite text-browser Lynks.

## Problem

Your text-browser will be given a set of graphic files and a simplified css-style-sheet. A graphic is defined by a name, height, width and a 2-dimensional array of characters. All characters are to be printed except for the character '`.`' which denotes a transparent pixel. Here is an example picture:

```
owl.png 5 7
.-----.
|O...O|
|..v..|
|.<_>.|
.-----.
```

Given the style-sheet your task is it to produce the graphical result that the browser is supposed to display. A CSS-file is made up from a number of entries where each entry looks like this:

```
#<id> {
 pos-x : <x> px ;
 pos-y : <y> px ;
 position : <relative = <id of graphic>|absolute> ;
 file : <filename> ;
 layer : <layer-number> ;
}
```

The following rules hold for the CSS-entries:

**Lines** Each CSS-entry will be given on exactly 7 lines as in the input above.

**Ordering** Each CSS-entry will contain exactly the 5 attributes `pos-x`, `pos-y`, `position`, `file` and `layer`, in this order, each attribute on a separate line.

**Whitespace** There may be zero or more white-spaces (spaces and tabs) at the beginning of lines, at the end of lines or everywhere where the sample above has a space.

Here are the rules for composing the picture:

**Background** The background is assumed to be black (i.e. just spaces).

**Positioning** The top left corner of the viewing device is assumed to be $x : 0$, $y : 0$. Absolute positioning always is based on this top-left corner. Relative positioning information is always based on the top-left pixel of another graphic. There will not be any circular references between CSS elements. All resulting positions will be zero or greater in $x$ and $y$.

**Layering** Graphics with a higher layer number are to be printed after graphics with a lower layer number. Graphics with the same layer number are to be printed in the order they appear in the CSS.

## Input

The first line of the input is the number of scenarios that will follow. For each scenario the following information is given: The first line contains the number of files to follow (at least one, at most 100), each of which is given by a space separated triple of a filename $f$, a height $h$, a width $w$ ($1 \leq w, h \leq 100$) and then $h$ lines, each with exactly $w$ characters. Following the file definition is a single line with a number $m$ (at least one, at most 500), which is followed by a CSS file of $m$ entries.

You can assume the resulting picture to be at most 1000 x 1000 characters large. All coordinates in CSS entries will be given as integers with an absolute value less than 1000000. All filenames and identifiers are made up from alphanumeric characters and dots only. No two files have the same name and no two identifiers are equal. The layer attribute will be at least 0 and at most 1000000.

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. For each scenario print the resulting picture from overlaying all the given graphics following the instructions in the CSS file. Your result for each scenario should be rectangular as small as possible. However, transparent pixels always belong to the resulting picture, even if they are located directly at the border. The top-left corner of the result should always contain position $(0, 0)$. All empty areas should be padded with spaces. Terminate the output for every scenario with a blank line.

## Sample Input

```
1
4
bg.png 5 7
.-----.
|.....|
|.....|
|.....|
.-----.
eye.jpg 1 1
O
nose.bmp 1 1
v
mouth.png 1 3
<_>
5
#bg {
pos-x: 1 px;
pos-y: 1 px;
position: absolute;
  file: bg.png;
  layer: 0;
}
#leftEye {
pos-x: 1 px;
pos-y: 1 px;
position: relative=bg;
  file: eye.jpg;
  layer: 1;
}
#rightEye {
pos-x: 4 px;
```

```
pos-y: 0 px;
position: relative=leftEye;
  file: eye.jpg;
  layer: 1;
}
#nose {
  pos-x: 2 px;
  pos-y: 1 px;
  position: relative=leftEye;
  file: nose.bmp;
  layer: 1;
}
#mouth {
  pos-x: -1 px;
  pos-y: 1 px;
  position: relative = nose;
  file: mouth.png;
  layer: 1;
}
```

## Sample Output

```
Scenario #1:

  -----
 |O   O|
 |  v  |
 | <_> |
  -----
```

# 9 Incomplete chess boards

## Background

Tom gets a riddle from his teacher showing 42 chess boards from each of which two squares are removed.

The teacher wants to know which boards can be completely covered by 31 dominoes. He promises ten bars of chocolate for the person who solves the problem correctly. Tom likes chocolate, but he cannot solve this problem on his own. So he asks his older brother John for help. John (who likes chocolate as well) agrees, provided that he will get half the prize.

John's abilities lie more in programming than in thinking and so decides to write a program. Can you help John? Unfortunately you will not win any bars of chocolate, but it might help you win this programming contest.

## Problem

You are given are 31 dominoes and a chess board of size $8 \times 8$, two distinct squares of which are removed from the board. The square in row $a$ and column $b$ is denoted by $(a, b)$ with $a, b \in \{1, \ldots, 8\}$.

A domino of size $2 \times 1$ can be placed horizontally or vertically onto the chess board, so it can cover either the two squares $\{(a, b),\ (a, b + 1)\}$ or $\{(b, a),\ (b + 1, a)\}$ with $a \in \{1, \ldots, 8\}$ and $b \in \{1, \ldots, 7\}$. The object is to determine if the so-modified chess board can be completely covered by 31 dominoes.

For example, it is possible to cover the board with 31 dominoes if the squares $(8, 4)$ and $(2, 5)$ are removed, as you can see in Figure 1.

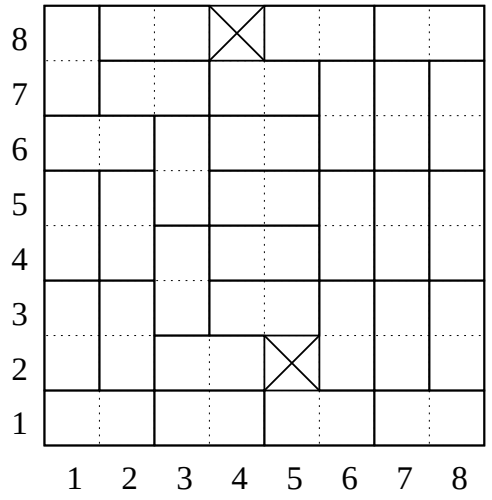



Figure 1: A possible covering where $(8, 4)$ and $(2, 5)$ are removed.

## Input

The first input line contains the number of scenarios $k$. Each of the following $k$ lines contains four integers $a$, $b$, $c$, and $d$, separated by single blanks. These integers in the range $\{1, \ldots, 8\}$ represent the chess board from which the squares $(a, b)$ and $(c, d)$ are removed. You may assume that $(a, b) \neq (c, d)$.

## Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. Then print the number `1` if the board in this scenario can be completely covered by 31 dominoes, otherwise write a `0`. Terminate the output of each scenario with a blank line.

## Sample Input

```
3
8 4 2 5
8 8 1 1
4 4 7 1
```

## Sample Output

```
Scenario #1:
1

Scenario #2:
0

Scenario #3:
0
```