ACM International Collegiate Programming Contest 2002–2003

Sponsored by COSYNUS, KOM, opus5 and sd&m

Darmstadt Regional Contest

Darmstadt University of Technology
Germany

June 29, 2002

PROBLEM SET

This problem set should contain 8 (eight) problems on 17 (seventeen) numbered pages. Please inform a runner immediately if something is missing from your problem set.

Sponsors

http://www.cosynus.de

http://www.kom.e-technik.tu-darmstadt.de/

http://www.opus5.de

http://www.sdm.de

Contents

1 World Cup Noise **3**

2 Warez Test **4**

3 Rubik's Cube **6**

4 Pumps and Pipes **9**

5 Self Numbers **11**

6 Strange Towers of Hanoi **12**

7 Darts **14**

8 Time Planner **16**

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).

Have fun!

1 World Cup Noise

Background

"KO-RE-A, KO-RE-A" shout 54.000 happy football fans after their team has reached the semifinals of the FIFA World Cup in their home country. But although their excitement is real, the Korean people are still very organized by nature. For example, they have organized huge trumpets (that sound like blowing a ship's horn) to support their team playing on the field. The fans want to keep the level of noise constant throughout the match.

The trumpets are operated by compressed gas. However, if you blow the trumpet for 2 seconds without stopping it will break. So when the trumpet makes noise, everything is okay, but in a pause of the trumpet, the fans must chant "KO-RE-A"!

Before the match, a group of fans gathers and decides on a chanting pattern. The pattern is a sequence of 0's and 1's which is interpreted in the following way: If the pattern shows a 1, the trumpet is blown. If it shows a 0, the fans chant "KO-RE-A". To ensure that the trumpet will not break, the pattern is not allowed to have two consecutive 1's in it.

Problem

Given a positive integer n, determine the number of different chanting patterns of this length, i.e., determine the number of n-bit sequences that contain no adjacent 1's. For example, for $n = 3$ the answer is 5 (sequences 000, 001, 010, 100, 101 are acceptable while 011, 110, 111 are not).

Input

The first line contains the number of scenarios.

For each scenario, you are given a single positive integer less than 45 on a line by itself.

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 n-bit sequences which have no adjacent 1's. Terminate the output for the scenario with a blank line.

Sample Input

```
2
3
1
```

Sample Output

```
Scenario #1:
5

Scenario #2:
2
```

2 Warez Test

Background

Our hero Jimmy is deeply in love with Christine and wants to marry her. So far so good, but Christine is the daughter of Mr. Warez, the owner of nearly all the warehouses in town. And Mr. Warez will never let his daughter marry someone who is not working for him. The sad point is that you may not work for Mr. Warez at all if you do not know how to push boxes in a warehouse *properly*. So Jimmy has to pass the infamous "Warez Test" first.

The Problem

Jimmy receives a number of maps each showing the outline of a warehouse. A map consists of squares. On one square there can be a wall or a single box or it can be empty. One or more squares are marked as target squares. One square is marked as the starting point for Jimmy.

Jimmy can move in four directions only: north, west, east and south. Jimmy can only move to an empty square or a square that is occupied by a box that can be pushed to the next square in the direction Jimmy is moving. This next square has to be empty which means that as a result of Jimmy moving one square at most one box can be moved exactly one square. By the way: Squares with a wall on it are not empty...

The border of every map (the first and last row and first and last column of the map) is always occupied by walls. On every map there are always as many boxes as there are target squares and there is always at least one box. It is Jimmy's task to find the shortest way to push the boxes onto the target squares so that at the end each box stands on a target square. To determine if one way is shorter than the other only the moves of Jimmy are counted, no matter whether he thereby pushes a box or not.

For every map it is guaranteed that there is exactly one shortest solution.

The Input

The first line contains the number of scenarios (maps).

For each map the first line contains the number of rows and columns of the map (both less than or equal to 15) separated by a blank. Then follows the map where '`X`' indicates a square with a wall on it, '`T`' indicates a target square and '`.`' indicates an empty square.

In the next line follows the starting position of Jimmy (row and column separated by a blank), then follows the number of boxes in the line after that. The following lines contain the starting positions of the boxes (one box per line, row and column separated by a blank).

The numbering of rows and columns starts with 0 in the upper left corner.

The Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting with 1. In the next line print the moves Jimmy has to make following the shortest possible solution. Print the letter '`n`' for Jimmy moving north (up), '`w`' for west (left), '`e`' for east (right) and '`s`' for south (down). The output for each scenario should be followed by a blank line.

(continued on next page)

Sample Input

```
2
8 6
XXXXXX
X.T..X
X....X
X....X
X....X
X....X
X....X
XXXXXX
5 3
1
4 3
5 4
XXXX
X.XX
X..X
XT.X
XXXX
1 1
1
2 1
```

Sample Output

```
Scenario #1:
nnnenw

Scenario #2:
s
```

3 Rubik's Cube

Background

Rummaging through the stuff of your childhood you find an old toy which you identify as the famous *Rubik's Cube*. While playing around with it you have to acknowledge that throughout the years your ability to solve the puzzle has not improved a bit. But because you always wanted to understand the thing and the only other thing you could do right now is to prepare for an exam, you decide to give it a try. Luckily the brother of your girlfriend is an expert and able to fix the cube no matter how messed-up it is. The problem is that he stays with his girlfriend in the Netherlands most of the time, so you need a solution for long-distance learning. You decide to implement a program which is able to document the state of the cube and the turns to be made.

Problem

A Rubik's Cube is covered with 54 square areas called *facelets*, 9 facelets on each of its six sides. Each facelet has a certain color. Usually when the cube is in its starting state, all facelets belonging to one side have the same color. For the original cube these are red, yellow, green, blue, white and orange.

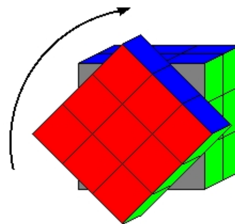

Figure 1: Turning the front side.

The positions of the facelets can be changed by turning the sides of the cube. This moves nine "little cubes" together with their attached facelets into a new position (see Fig. 1).

The problem is to determine how the facelets of the entire cube are colored after turning different sides in different directions.

Input

The first line contains the number of scenarios. Each scenario consists of two sections. The first section describes the starting state of the cube and the second describes the turns to be made.

The starting state describes the colors of the facelets and where they are positioned. The colors are identified by single characters, and one character is given per facelet. Characters are separated by blanks and arranged in a certain pattern (see Fig. 2). The pattern identifies all six sides of the cube and can be thought of as a folding pattern. As shown in Fig. 2, the description of the top side of the cube is placed right over the description of the front side. This is done by indenting the lines with blanks. The next three lines contain the descriptions of the left, front, right and back side as shown in Fig. 2. The descriptions are simply concatenated with a blank character used as separator. After that the description of the bottom side follows, using the same format as the one used to describe the top side. This concludes the description of the starting state.

Then follows the second section of the scenario containing the turns which have to be performed. The description of the turns starts with a line containing the number of turns t ($t > 0$). Each turn is given in a separate line and consists of two integer values s and d which are separated by a single blank. The first value s determines the side of the cube which has to be turned. The sides are serially numbered as follows: left '0', front '1', right '2', back '3', top '4', bottom '5'. The second value d determines in which direction

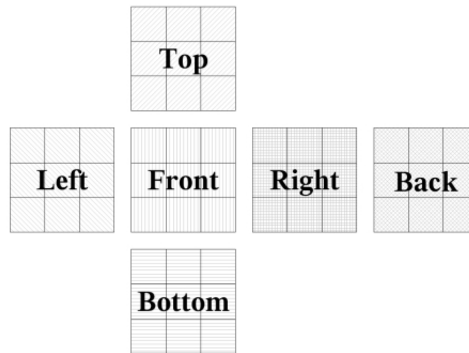

Figure 2: Folding pattern for input and output.

the side s has to be turned and can either be '1' or '-1'. A '1' stands for clockwise and a '-1' for counter-clockwise. The direction is given under the assumption that the viewer is looking directly at the specific side of the cube.

Output

The output for every scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. After this line print the resulting state of the cube using the same format as the input. Each scenario is terminated by a single blank line.

Sample Input

```
2
      w w w
      w w w
      w w w
r r r g g g b b b o o o
r r r g g g b b b o o o
r r r g g g b b b o o o
      y y y
      y y y
      y y y
2
3 1
0 -1
      g b b
      g w w
      g w w
r r r y g g b b y o o w
r r r y g g b b y o o w
w w w r g g b b y o o b
      o y y
      o y y
      o r r
2
0 1
3 -1
```

Sample Output

```
Scenario #1:
      g b b
      g w w
      g w w
r r r y g g b b y o o w
r r r y g g b b y o o w
w w w r g g b b y o o b
      o y y
      o y y
      o r r

Scenario #2:
      w w w
      w w w
      w w w
r r r g g g b b b o o o
r r r g g g b b b o o o
r r r g g g b b b o o o
      y y y
      y y y
      y y y
```

4 Pumps and Pipes

Background

For hundreds of years fire departments all over the world have been using water for their fight against the fire. Unfortunately, one cannot always find enough water right where the fire is burning. Accordingly, the fire departments are equipped with lots of pumps and pipes to transport the water to where it is needed. Setting up the system of pumps and pipes might not be such an easy task, as there are several restrictions one has to take care of.

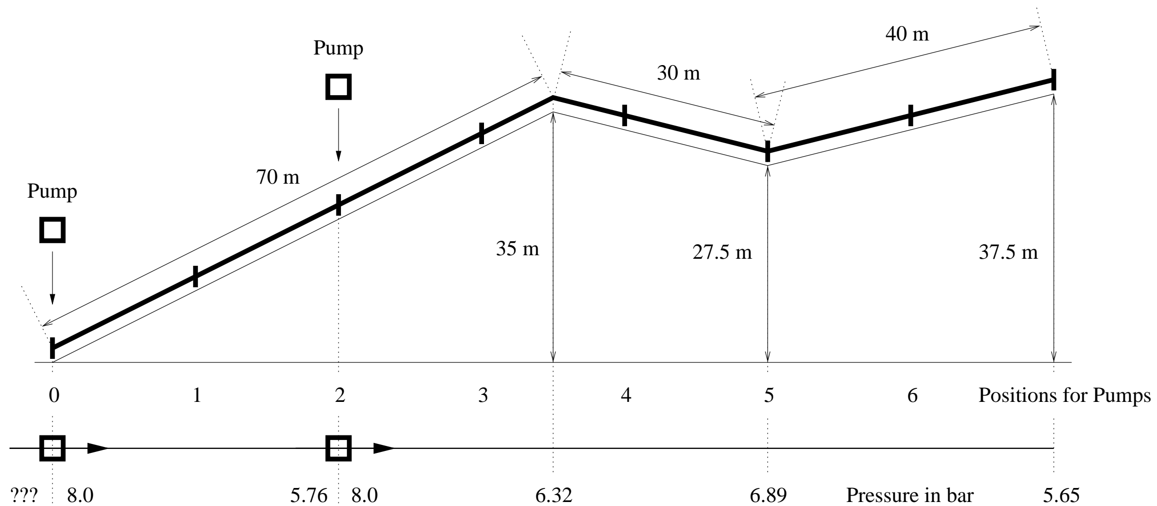

Figure 3: First Scenario in Sample Input

The Problem

Let us assume that only one type of pipes is used for the transportation of water, namely pipes with a diameter of 75 millimeters and a length of 20 meters. Depending on the quantity of water that is pumped through the tubes, there is a loss of pressure per meter according to the following table:

flow (f) in liters per minute	loss of pressure in millibar per meter
200	1
400	3
600	6
800	10
1000	15
1200	20

The pressure is also affected by the profile of the landscape, more precisely the pressure changes by 1 bar every 10 meters of vertical distance, decreasing for pipes running uphill and increasing for pipes running downhill. Pumps need an input pressure of at least 2 bar and produce a constant output pressure of 8 bar, but they cannot be used to reduce pressure. Pipes do not resist a pressure higher than 12 bar, and for a constant flow a pressure of at least 2 bar is needed at all points. At the end of the line, the pressure must be at least 5 bar and at most 8 bar to permit effective fire fighting. There is always a pump at the beginning of the line of pipes (position 0). Other pumps can be placed wherever two pipes are attached to each other but not at the end.

Write a program to find the least number of pumps necessary, and their positions. If several solutions with the least number of pumps exist, prefer the ones where pumps are placed closer to the beginning of the line (carrying these pumps is no fun...).

The Input

The first line contains the number of scenarios.

For each scenario, you are first given an integer $f \in \{200, 400, 600, 800, 1000, 1200\}$ on a line by itself, indicating the desired flow in liters per minute. The next line contains two integers n and m, separated by a single blank, where $1 \leq n \leq 20$ is the number of pipes to be used, and $1 \leq m \leq 400$ is the number of segments with a constant slope.

The following m lines describe these m segments, each containing integers l and s, separated by a single blank, where $l > 0$ is the length in meters and s is the slope measured in per cent ($s = 10$ means that pipes of a length of 100 meters ascend by 10 meters, $s = -50$ means they descend 50 meters; $-100 \leq s \leq 100$). It is guaranteed that the m given lengths add up to n times 20 meters.

The Output

The output for each scenario begins with a line containing "`Scenario #i:`", where `i` is the number of the scenario starting at 1. The next line contains the number of pumps in the optimal solution (if it exists), followed by a colon "`:`" and a single blank, and the positions of the pumps separated by commas "`,`" and no blanks. If no placement of the pumps satisfies the given criteria, print a line containing "`no solution`" instead. Finish the output of each scenario with an additional blank line.

Sample Input

```
2
600
7 3
70 50
30 -25
40 25
1000
8 4
20 0
80 -100
20 10
40 30
```

Sample Output

```
Scenario #1:
2: 0,2

Scenario #2:
no solution
```

5 Self Numbers

Background

In 1949 the Indian mathematician D.R. Kaprekar discovered a class of numbers called *self-numbers*. For any positive integer n, define $d(n)$ to be n plus the sum of the digits of n. (The d stands for *digitadition*, a term coined by Kaprekar.) For example:

$$d(75) = 75 + 7 + 5 = 87$$

Given any positive integer n as a starting point, you can construct the infinite increasing sequence of integers $n, d(n), d(d(n)), d(d(d(n))), \ldots$ For example, if you start with 33, the next number is $33 + 3 + 3 = 39$, the next is $39 + 3 + 9 = 51$, the next is $51 + 5 + 1 = 57$, and so you generate the sequence

$$33, 39, 51, 57, 69, 84, 96, 111, 114, 120, 123, 129, 141, \ldots$$

The number n is called a *generator* of $d(n)$. In the sequence above, 33 is a generator of 39, 39 is a generator of 51, 51 is a generator of 57, and so on.

Some numbers have more than one generator: For example, 101 has two generators, 91 and 100. A number with no generators is a *self-number*. There are thirteen self-numbers less than 100: 1, 3, 5, 7, 9, 20, 31, 42, 53, 64, 75, 86, and 97.

Problem

Write a program to output all positive self-numbers less than 10000 in increasing order, one per line.

Input

There is no input.

Output

All positive self-numbers less than 10000 in increasing order, one per line.

6 Strange Towers of Hanoi

Background

Charlie Darkbrown sits in another one of those boring Computer Science lessons: At the moment the teacher just explains the standard *Tower of Hanoi* problem, which bores Charlie to death!

Figure 4: The standard (three) Towers of Hanoi.

The teacher points to the blackboard (Fig. 4) and says: “So here is the problem:

- There are three towers: A, B and C.
- There are n disks. The number n is constant while working the puzzle.
- All disks are different in size.
- The disks are initially stacked on tower A increasing in size from the top to the bottom.
- The goal of the puzzle is to transfer all of the disks from tower A to tower C.
- One disk at a time can be moved from the top of a tower either to an empty tower or to a tower with a larger disk on the top.

So your task is to write a program that calculates the smallest number of disk moves necessary to move all the disks from tower A to C.”

Charlie: “This is incredibly boring — everybody knows that this can be solved using a simple recursion. I deny to code something as simple as this!”

The teacher sighs: “Well, Charlie, let’s think about something for *you* to do: For you there is a fourth tower D. Calculate the smallest number of disk moves to move all the disks from tower A to tower D using all four towers.”

Charlie looks irritated: “Urgh... Well, I don’t know an optimal algorithm for four towers...”

Problem

So the real problem is that problem solving does not belong to the things Charlie is good at. Actually, the only thing Charlie is really good at is “sitting next to someone who can do the job”. And now guess what — exactly! It is you who is sitting next to Charlie, and he is already glaring at you.

Luckily, you know that the following algorithm works for $n \leq 12$: At first $k \geq 1$ disks on tower A are fixed and the remaining $n-k$ disks are moved from tower A to tower B using the algorithm for four towers. Then the remaining k disks from tower A are moved to tower D using the algorithm for three towers. At last the $n-k$ disks from tower B are moved to tower D again using the algorithm for four towers (and thereby not moving any of the k disks already on tower D). Do this for all $k \in \{1, \ldots, n\}$ and find the k with the minimal number of moves.

So for $n = 3$ and $k = 2$ you would first move 1 $(3-2)$ disk from tower A to tower B using the algorithm for four towers (one move). Then you would move the remaining two disks from tower A to tower D using the algorithm for three towers (three moves). And the last step would be to move the disk from tower B to tower D using again the algorithm for four towers (another move). Thus the solution for $n = 3$ and $k = 2$ is 5 moves. To be sure that this really is the best solution for $n = 3$ you need to check the other possible values 1 and 3 for k. (But, by the way, 5 is optimal...)

Input

There is no input.

Output

For each n ($1 \leq n \leq 12$) print a single line containing the minimum number of moves to solve the problem for four towers and n disks.

7 Darts

Background

Many nations (including Germany) have a strange tradition of throwing small arrows at round flat targets (usually, these small arrows are called *darts* and so is the game).

In a darts game, the target consists of a flat circle which is divided into slices and rings. The slices are numbered from 1 to 20 and the rings are called *double* or *treble ring* (see Figure 5). The center part of the board is called the *bull's eye* which is further subdivided into an inner part (the real bull's eye) and an outer part (called the *bull*, see Fig. 5).

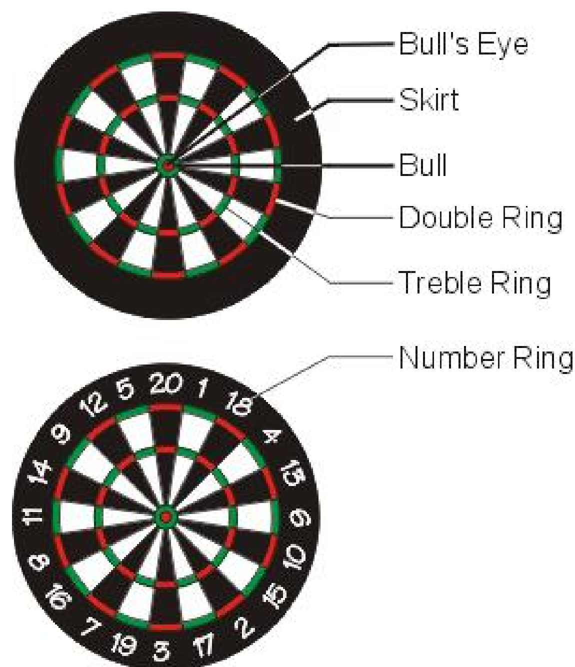

Figure 5: Layout of a dart board.

Players take turns in throwing the darts at the board. Their score depends on the areas they hit with their darts. Hitting the 20 slice in the double ring scores $2 \cdot 20 = 40$ points. Hitting the treble ring multiplies the score by 3. The inner part of the bull's eye counts 50, the outer part 25 points.

Every turn consists of 3 darts being thrown at the dartboard by a player and his score is the sum of the scores of all darts which hit the dartboard in one of the numbered areas.

Problem

Your friends have played darts yesterday and from their match the scores are still on the blackboard in your room. From reading the scores, you would like to know, how the individual players threw their darts and where they could have hit the dartboard. You are to write a program which, given the score of a turn, reconstructs the number of possible distinct combinations of hits of the three darts on the dartboard ignoring the order in which the darts are thrown.

As an example, consider the overall score of 3 of a player. This could have happened as follows:

3 =	0 +	0 +	1·3	one dart hits slice 3
3 =	0 +	0 +	3·1	one dart hits slice 1 in treble ring
3 =	0 +	1·1 +	1·2	one dart hits slice 1 and one dart hits slice 2
3 =	0 +	1·1 +	2·1	one dart hits slice 1 and one dart hits slice 1 in double ring
3 =	1·1 +	1·1 +	1·1	all three darts hit slice 1

The resulting sum of possible distinct combinations is 5.

A more complex example is score 9:

9 =	0 +	0 +	$1 \cdot 9$	one dart hits slice 9
9 =	0 +	0 +	$3 \cdot 3$	one dart hits slice 3 in treble ring
9 =	0 +	$1 \cdot 1$ +	$1 \cdot 8$	one dart hits slice 1 and one dart hits slice 8
9 =	0 +	$1 \cdot 1$ +	$2 \cdot 4$	one dart hits slice 1 and one dart hits slice 4 in double ring
⋮				
9 =	0 +	$3 \cdot 2$ +	$1 \cdot 3$	one dart hits slice 2 in treble ring and one dart hits slice 3
9 =	$1 \cdot 1$ +	$1 \cdot 1$ +	$1 \cdot 7$	two darts hit slice 1 and one dart hits slice 7
⋮				
9 =	$2 \cdot 1$ +	$3 \cdot 1$ +	$2 \cdot 2$	one dart hits slice 1 in double ring, one dart hits slice 1 in treble ring and one dart hits slice 2 in double ring
9 =	$1 \cdot 3$ +	$1 \cdot 3$ +	$1 \cdot 3$	three darts hit slice 3
9 =	$1 \cdot 3$ +	$1 \cdot 3$ +	$3 \cdot 1$	two darts hit slice 3 and one dart hits slice 1 in treble ring
9 =	$1 \cdot 3$ +	$3 \cdot 1$ +	$3 \cdot 1$	one dart hits slice 3 and two darts hit slice 1 in treble ring
9 =	$3 \cdot 1$ +	$3 \cdot 1$ +	$3 \cdot 1$	three darts hit slice 1 in treble ring

What is the number of combinations? Write a program to find out.

Input

The first line contains the number of scenarios.

For each scenario, you are give a dart score as a single positive integer on a line by itself.

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 of possible dart score combinations on a line by itself. Finish the output of every scenario with a blank line.

Sample Input

```
2
3
9
```

Sample Output

```
Scenario #1:
5

Scenario #2:
41
```

8 Time Planner

Background

Problems in computer science are often classified as belonging to a certain class of problems (e.g. NP, NP complete, unsolvable, ...). Whatever class of problems you have to solve in a team, there is always one *main* problem: To find a date where all the programmers can meet to work together on their project.

Problem

Your task is to write a program that finds all possible appointments for a programming team that is busy all the time and therefore has problems in finding accurate dates and times for their meetings.

Input

The first line of the input contains the number of scenarios.

For each scenario, you are first given the number m of team members in a single line, with $2 \leq m \leq 20$. For every team member, there will be a line containing the number n of entries in this members calender ($0 \leq n \leq 100$), followed by n lines in the following format:

```
YYYY MM DD hh mm ss YYYY MM DD hh mm ss some string here
```

Each such line contains the year, month, day, hour, minute, and second of both starting and ending time for the appointment, and a string with the description of the appointment. All numbers are zero padded to match the given format, with single blanks in-between. The string at the end might contain blanks and is no longer than 100 characters. The dates are in the range January 1, 1800 midnight to January 1, 2200 midnight. For simplification, you may assume that all months (even February) have 30 days each, and no invalid dates (like January 31) will occur.

Note that the ending time of a team member's appointment specifies the point in time where this team member is ready to join a meeting.

Output

The output for each scenario begins with a line containing `Scenario #i:`, where `i` is the number of the scenario starting at 1. Then print a line for each possible appointments that follows these rules:

- at any time during the meeting, at least two team members need to be there
- at any time during the meeting, at most one team member is allowed to be absent
- the meeting should be at least one hour in length
- all team members are willing to work 24 hours a day

For example, if there are three team members A, B and C, the following is a valid meeting: It begins solely with A and B. Later C joins the meeting and before the end, A may leave.

Always print the longest appointment possible satisfying these conditions, even if it is as long as 400 years. Sort these lines by date and time, and use the following format:

```
appointment possible from MM/DD/YYYY hh:mm:ss to MM/DD/YYYY hh:mm:ss
```

The numbers indicating month, day, year, hour, minute, and second should be zero padded in order to get the required number of characters. If no appointment is possible, just print a line containing

```
no appointment possible
```

Conclude the output for each scenario with a blank line.

Sample Input

```
2
3
3
2002 06 28 15 00 00 2002 06 28 18 00 00 TUD Contest Practice Session
2002 06 29 10 00 00 2002 06 29 15 00 00 TUD Contest
2002 11 15 15 00 00 2002 11 17 23 00 00 NWERC Delft
4
2002 06 25 13 30 00 2002 06 25 15 30 00 FIFA World Cup Semifinal I
2002 06 26 13 30 00 2002 06 26 15 30 00 FIFA World Cup Semifinal II
2002 06 29 13 00 00 2002 06 29 15 00 00 FIFA World Cup Third Place
2002 06 30 13 00 00 2002 06 30 15 00 00 FIFA World Cup Final
1
2002 06 01 00 00 00 2002 06 29 18 00 00 Preparation of Problem Set
2
1
1800 01 01 00 00 00 2200 01 01 00 00 00 Solving Problem 8
0
```

Sample Output

```
Scenario #1:
appointment possible from 01/01/1800 00:00:00 to 06/25/2002 13:30:00
appointment possible from 06/25/2002 15:30:00 to 06/26/2002 13:30:00
appointment possible from 06/26/2002 15:30:00 to 06/28/2002 15:00:00
appointment possible from 06/28/2002 18:00:00 to 06/29/2002 10:00:00
appointment possible from 06/29/2002 15:00:00 to 01/01/2200 00:00:00

Scenario #2:
no appointment possible
```