Problem A. A Constant Struggle

Source file name:	constant.c, constant.cpp, constant.java, constant.py
Input:	`standard`
Output:	`standard`

Your math teacher Xavier Guha has given your class an extra credit assignment to work on. The problem he gives is as follows: Given a linear equation of the form

$$c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_6x_6 + c_7x_7 + c_8x_8 = N$$

with $c_1, \ldots, c_8, N$ given, he asks you to give the unique solution x (a vector of length 8) to the equation, where each x_i is a non-negative integer. Of course, being a clever student, you realize that depending on the values of c and N, there may be no solution or there may be several solutions. After class, you approach him and inform him of the mistake, but he is stubborn and will not have any of your nonsense.

Having taken several programming classes from his brother, you decide to prove your teacher wrong[1] by writing a program to determine how many solutions the equation has.

Given $c_1, \ldots, c_8$ and N, determine how many unique solutions the equation has. Two solutions p and q are considered unique (different) if there exists some i for which $p_i \neq q_i$. Since you may get this assignment again in the future, your program should be able to solve several instances of the equation.

Input

Input will begin with a positive integer T denoting the number of equations to solve. This will be followed by T lines, each containing an instance of the equation to solve. Each instance will be described by 9 space separated positive integers, all ≤ 100. The first 8 numbers represent $c_1, \ldots, c_8$, and the 9^{th} number represents N.

Output

For each equation, output a single line `Equation #E: S` where E is the equation number beginning with 1 and S is the number of unique solutions to the equation. It is guaranteed that the value of S will fit in a 64-bit signed integer.

Example

Input	Output

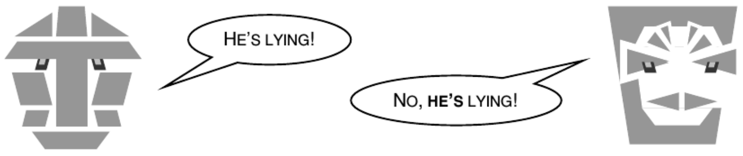
`5`
`1 1 1 1 1 1 1 1 1`
`1 2 3 4 5 6 7 8 9`
`2 4 6 8 10 12 14 16 29`
`2 4 6 8 10 12 14 17 17`
`1 1 1 1 1 1 1 1 100`
	`Equation #1: 8`
	`Equation #2: 29`
	`Equation #3: 0`
	`Equation #4: 1`
	`Equation #5: 26075972546`

[1]This (proving your teacher is wrong) is generally a bad idea. The author of this problem absolves himself of any (liability for) ill will created between you and your teacher, as well as any detrimental effect this may have on your final course grade. Solve at your own risk.

Problem B. Tautobots and Contradicticons

Source file name:	logotron.c, logotron.cpp, logotron.java, logotron.py
Input:	`standard`
Output:	`standard`

The planet Logotron is inhabited by two types of robots - the Tautobots and the Contradicticons. The Tautobots are programmed to always tell the truth, while the Contradicticons must always lie. Unfortunately, there is no simple way for outsiders to tell them apart, which often causes problems.



You are given a set of statements made by a group of robots. Every robot knows the type of every other robot, as well as itself. Each statement consists of an author (the robot that made the statement), a subject (the robot the statement is about), and the type of the subject (Tautobot or Contradicticon). For example, "Robot 5 says that Robot 2 is a Tautobot" is a valid statement. Note that if Robot 5 is a Contradicticon, then Robot 2 must also be a Contradicticon, since Robot 5 lied.

Given M statements made by N robots, you must find the number of distinct ways to assign a type to each robot, consistent with the statements. Two assignments are considered to be different if at least one robot is a Tautobot in one and a Contradicticon in the other.

Input

The first input line contains a positive integer T, indicating the number of test cases to be processed. This will be followed by T test cases.

Each test case is formatted as follows. The first line consists of the numbers N ($1 \leq N \leq 15$) and M ($0 \leq M \leq 100$). This is followed by M lines, each of which represents a statement by one of the robots. A statement is formatted as "A S X". Here A and S are integers between 1 and N (inclusive) representing the author of the statement and its subject respectively (assume that A and S will be different robots). X will be one of the characters 'T' (for Tautobot) or 'C' (for Contradicticon).

Assume that a robot will not contradict itself (making a statement and then making the opposite of that statement) but different robots may contradict each other (in that case, there is no possible answer, i.e., zero assignments). Also assume that we will not have the same statement repeated by a robot several times, i.e., no two input statements will be completely identical.

Output

For each test case, output a single line, formatted as: `Case #t:`, where t is the test case number (starting from 1), a single space, and then the number of distinct assignments that can be made for that case.

Example

Input	Output
3	Case #1: 2
3 2	Case #2: 4
1 2 T	Case #3: 4
2 3 C	
4 2	
1 2 T	
2 3 C	
2 0	

Problem C. Chocolate Fix

Source file name:	chocolate.c, chocolate.cpp, chocolate.java, chocolate.py
Input:	`standard`
Output:	`standard`

Your cousin recently bought a game that involves arranging truffles on a 3×3 grid. Eager to one up him, you decide to write a program to solve the puzzles automatically, so that he doesn't have a chance in figuring out the puzzles against you!

The game comes with nine truffles, which are used in each puzzle. Each truffle is either vanilla, strawberry or chocolate (VSC) and the shape of each truffle is either square, round or triangular (SRT). There is exactly one truffle of each combination of attributes and all of them must be used. In each puzzle, you must place each of the nine truffles on the 3×3 board. For convenience, label the squares 1, 2 and 3 on the first row, from left to right, 4, 5 and 6 on the second row, and 7, 8 and 9 on the third row as illustrated below:

1	2	3
4	5	6
7	8	9

For each puzzle, you are given a list of clues. Each clue represents partial information about a subsection of the board. For example, the clue below means that there is some 3×2 window with the square strawberry truffle in its upper right corner:

```
__ SS
-- --
-- --
```

There are only two such possible 3×2 windows on a 3×3 board. Thus, this clue essentially conveys the fact that the square strawberry truffle must appear in either square 2 or square 3 of the board since clues cannot be rotated:

__	SS	
--	--	
--	--	

	__	SS
	--	--
	--	--

Another example of a clue is as follows:

```
__ _S
__ RC
__ T_
```

This clue also represents a 3×2 window of the board. It indicates that in the right column of the window there must be a strawberry truffle (of some shape) on the top row, the round chocolate truffle in the middle row of the column and a triangular truffle (of some flavor) on the bottom row of the column. Due to the window size, we can ascertain that this must be true of either middle or rightmost column of the game board.

Given a set of clues that uniquely specifies a solution to a chocolate puzzle, determine that solution.

Input

The first input line contains a positive integer, n, indicating the number of puzzles to solve. The puzzles follow. The first line of each puzzle will contain a single positive integer, c ($1 \leq c \leq 10$), representing the number of clues for that puzzle. The clues will follow. The first line of each clue will contain two space separated integers: x ($1 \leq x \leq 3$) and y ($1 \leq y \leq 3$), representing the number of rows and columns, respectively, for the clue window. The next x lines will contain the clue, with each element in the clue represented by 2 characters. The first character will come from the set {'_', 'S', 'R', 'T'}, indicating the shape of that element and the second character will come from the set {'_', 'V', 'S', 'C'}, indicating the flavor of that element. Note that the underscore character simply means that that attribute is not fixed. Each of these x lines will contain y codes; the codes separated by exactly one space.

Output

The first line of the output for each puzzle should be `Puzzle #p:`, where p is the puzzle number, starting with 1. On the next three lines, output the puzzle solution, using the same codes used in the clues. Note that since there is a unique solution to each puzzle, no underscore characters can be in any valid solution.

Example

Input

```
3
4
3 3
TC __ SS
__ __ __
__ TV __
2 3
__ SC __
RV __ SV
3 3
__ __ __
__ RC __
__ __ __
2 3
__ __ __
TS __ RS
5
2 3
__ __ __
__ __ RC
2 2
__ RS
SC __
2 2
SV TC
__ __
3 2
TV __
__ __
__ RV
3 2
__ TS
__ __
__ __
3
3 2
_C R_
_C __
S_ _C
1 2
TC _V
3 2
_V __
S_ S_
T_ _V
```

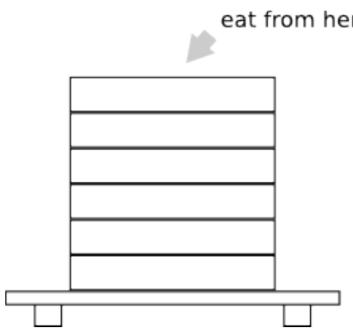
Output

```
Puzzle #1:
TC SC SS
RV RC SV
TS TV RS
Puzzle #2:
TV RS TS
SC SV TC
SS RV RC
Puzzle #3:
TV TC RV
SS SC RS
TS SV RC
```

Problem D. Dirty Plates

Source file name:	dirty.c, dirty.cpp, dirty.java, dirty.py
Input:	`standard`
Output:	`standard`

Lazy Louie loves to eat but he adamantly hates cleaning. That is why he has chosen to delay cleaning for as long as possible. Being an avid inventor, Louie created an invention to help increase his laziness. That invention is the two sided plate! Two sided plates can be stacked like coins for easy storage. They also have the interesting feature that you may eat from either side of the plate.



Lazy Louie lacks cabinet space. That is why he stores his two sided plates directly on his dining table. When enjoying a meal he simply eats off the topmost plate, although he will only eat off the topmost surface if it is clean. When the plates are stacked, if a dirty side of the plate touches the clean side of another plate then the grime of that plate transfers to the clean plate making that side of the plate dirty. The side that was originally dirty stays dirty.

Since Louie is stacking plates on his table, if a dirty side of a plate touches the table then the grime will transfer to the table making the table dirty. If the table is dirty already and it touches a clean side of a plate then that plate's side becomes dirty. Note that the table remains dirty. Also, both sides remain dirty when two dirty sides, either the table or plates, touch.

Louie would like to know the maximum number of meals that can be eaten before any cleaning is done. Louie is given the number of plates he already has of three kinds: plates that are clean on both sides, plates that are clean on one side while dirty on the other, and plates that are dirty on both sides. Before eating his first meal he can stack these plates in any way he likes on the table. After eating each meal the topmost plate becomes dirty. Between meals Louie can rearrange the plates in any way he likes. Louie is allowed to change the order of the plates on the stack and change which side is facing up but they must remain a stack of plates after being rearranged.

Input

The first input line contains a positive integer, n, indicating the number of eating scenarios to analyze. The next n lines contain the description of the plates. Each line contains three integers, c, s, and d,

($0 \leq c,\ s,\ d \leq 100$) representing (respectively) the number of plates with both sides clean, the number of plates with one side clean and the number of completely dirty plates.

Output

For each scenario, first output the heading `Scenario #d:`, where d is the scenario number, starting with 1. Then, print the maximum number of times a meal can be eaten before Louie has to clean.

Example

Input	Output

4
1 0 0
2 0 0
1 1 3
2 2 2
	Scenario #1: 2
	Scenario #2: 3
	Scenario #3: 2
	Scenario #4: 4

Problem E. Wheel of Universally Copious Fortune

Source file name:	copious.c, copious.cpp, copious.java, copious.py
Input:	`standard`
Output:	`standard`

In the game "Wheel of Fortune", the number of letters in a word is given and the contestants guess the letters in the word and, as some letters appear, the contestants guess the word. But, you are a computer scientist and know that you can write a program to search a dictionary and provide candidate words (possible matches) for you.

Given the dictionary and a partially-defined word, you are to determine the candidate words. Note that there may be no candidate words for a given partially-defined word.

Input

The first input line contains an integer n ($1 \leq n \leq 100$), indicating the number of words in the dictionary. The dictionary words will be on the following n input lines, one word per line. Each word starts in column 1, contains only lowercase letters, and will be 1-20 letters (inclusive). Assume that the dictionary words are distinct, i.e., no duplicates. The next input line will contain a positive integer m, indicating the number of words to be checked against the dictionary. These words will be on the following m input lines, one word per line. Each word starts in column 1, contains only lowercase letters and hyphens, and will be 1-20 characters (inclusive). A letter in a position indicates that the word must have that letter in that position; a hyphen in a position indicates that any letter can be in that position.

Output

At the beginning of each word to be checked, output `Word #w:`, where w is the word number (starting from 1). Then print the input word to be checked. Then, on the following output lines, print the candidate words from dictionary that could be a match (print these words in the order they appear in the dictionary). Also print the total number of candidate words (possible matches).

Follow the format illustrated in Example Output.

Example

Input	Output
8 at cat ali sat nerds coach couch ninja 5 co-ch -at --- ali a-c	Word #1: co-ch coach couch Total number of candidate words = 2 Word #2: -at cat sat Total number of candidate words = 2 Word #3: --- cat ali sat Total number of candidate words = 3 Word #4: ali ali Total number of candidate words = 1 Word #5: a-c Total number of candidate words = 0

Problem F. Factorial Products

Source file name:	fact.c, fact.cpp, fact.java, fact.py
Input:	`standard`
Output:	`standard`

Factorial is just a game of multiplications. Formally, it can be defined as a recurrence relation:

Fact (0) = 1
Fact (n) = n * Fact(n-1), for all integers n > 0

This problem is all about multiplications, more and more multiplications. It is a game of multiplications of factorials.

You will be given three lists of numbers: A, B and C. You have to take the factorials of all the numbers in each list and multiply them to get `ProFact(A)`, `ProFact(B)`, `ProFact(C)`. Then report which product is the largest.

For example, consider the lists $A = \{2, 4, 7\}$, $B = \{0, 1, 9\}$ and $C = \{2, 3, 5, 5\}$. Then,

```
ProFact(A) = 2! * 4! * 7! = 241920
ProFact(B) = 0! * 1! * 9! = 362880
ProFact(C) = 2! * 3! * 5! * 5! = 172800
```

So, the largest product for this example is `ProFact(B)`.

Input

The first input line contains a positive integer, n, indicating the number of test cases. Each test case consists of four input lines. The first line consists of three positive integers providing, respectively, the size for the lists A, B and C. The last three lines contain, respectively, the elements (non-negative integers) in lists A, B and C.

All the values in the input file will be less than 2501.

Output

For each test case, output `Case #t: h` in a line, where t is the case number (starting with 1) and h is the list name with the highest product. If two or three lists are tied for the highest product, print `TIE`. Follow the format illustrated in Example Output.

Assume that, if the pairwise product values differ, then the relative difference of these products will differ by at least 0.01% of the largest product.

Example

Input	Output

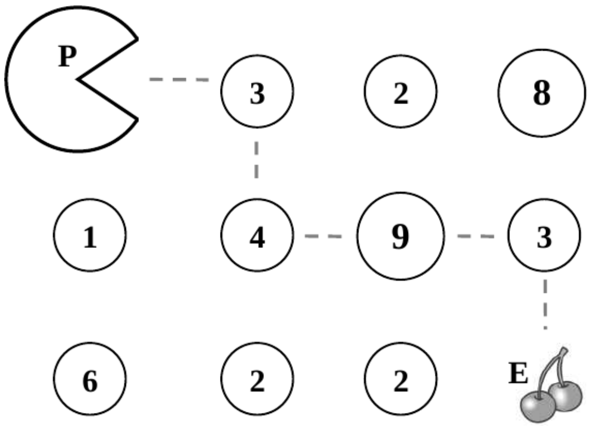
3
3 3 4
2 4 7
0 1 9
2 3 5 5
2 2 2
2 3
3 2
2 2
3 3 3
1 3 5
2 4 6
1 4 7
	Case #1: B
	Case #2: TIE
	Case #3: C

Problem G. Pac Man for your New Phone

Source file name:	pacman.c, pacman.cpp, pacman.java, pacman.py
Input:	`standard`
Output:	`standard`

You are writing an app for your friend's new Phone, the newPhone. Since you grew up on Pac Man, you want to write a simplified version of the game. In this game, the board is a rectangular grid and Pac Man starts at the upper left-hand corner. His goal is to get to the lower right-hand corner. He always moves one square to the right or one square down. Each square he goes to has a "goody" that's worth a particular amount of points. Your score is simply the sum of the scores of the goodies in each square you have visited.

For example, if the game board looks like this (P indicates Pac Man's starting location, and E indicates his ending location):



then Pac Man's optimal strategy would be to move right, down, right, right again, then down to yield a score of $3 + 4 + 9 + 3 = 19$.

Given a game board, determine the maximum possible score for Pac Man.

Input

There will be multiple game boards in the input. The first input line contains a positive integer n, indicating the number of game boards to be processed. The first line of each game board will contain two positive integers, r $(0 < r < 100)$ and c $(1 < c < 100)$, representing the number of rows and columns for this game board. (The example above has three rows and four columns.) Each of the following r input lines will contain c tokens, representing the contents of that row. The first item on the first of these lines will be the character 'P', representing Pac Man's original location and the last item on the last line will be the character 'E', representing Pac Man's goal location. The rest of the items will be positive integers less than 1000. Items will be separated by a single space on each line.

Output

For each test case, print a single line with `Game Board #g: s`, where g is the input board number (starting from 1) and s is the maximum possible score for the game.

Example

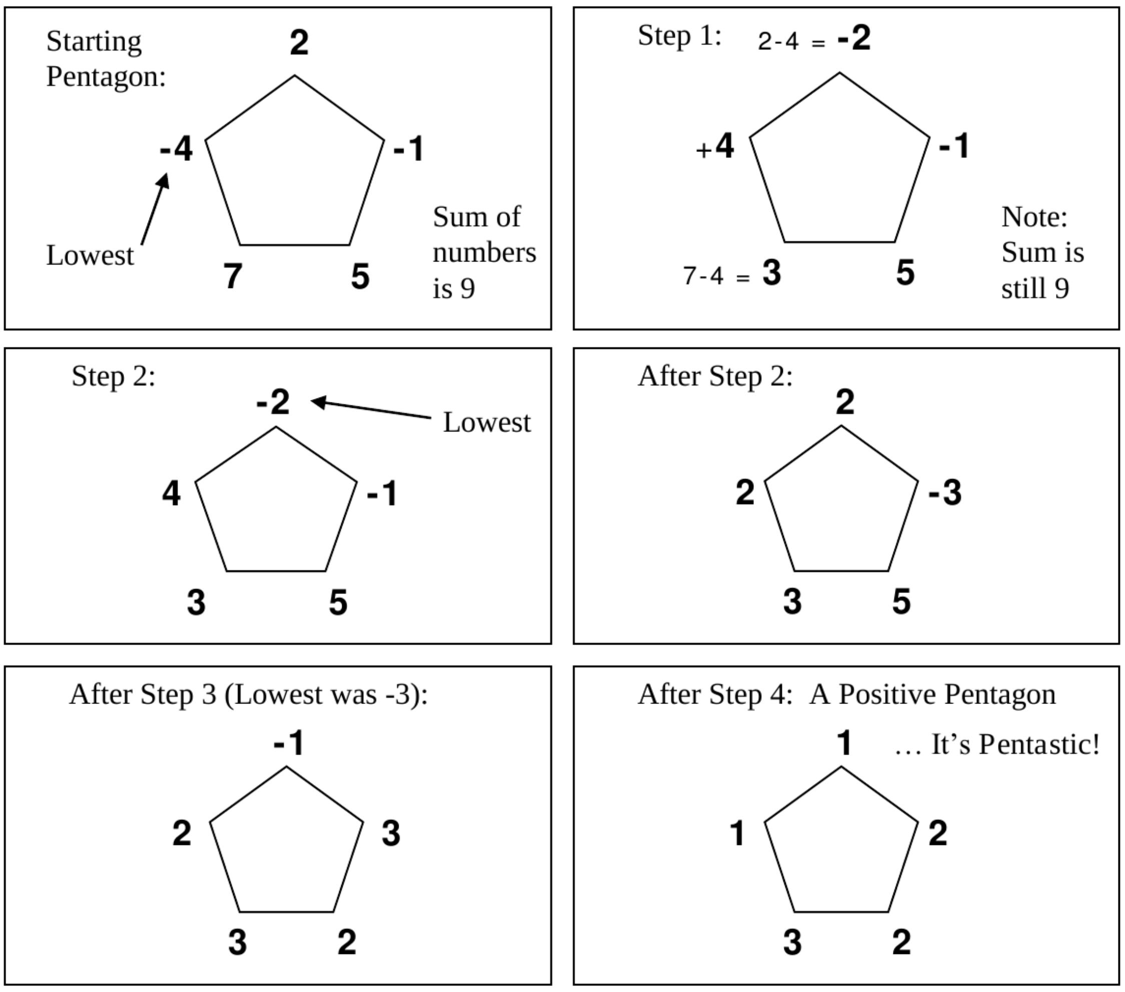
Input	Output
2 3 4 P 3 2 8 1 4 9 3 6 2 2 E 2 2 P 5 401 E	Game Board #1: 19 Game Board #2: 401

Problem H. Positively Pentastic!

Source file name:	pentastic.c, pentastic.cpp, pentastic.java, pentastic.py
Input:	`standard`
Output:	`standard`

Five random integers are placed at the corners of a pentagon. Typically, some of these numbers will be negative, but their sum is guaranteed to be positive. The goal is to get rid of all the negative numbers through a balanced process of subtraction and negation.

Starting with the lowest of the negative numbers, we negate the number (thus making it positive), and then subtract that value from each of its two neighbors. The sum of the new numbers will remain the same as the original pentagon, so the pentagon is still "balanced." This process (finding the lowest of the negative numbers, negating it, and subtracting it from its neighbors) is then repeated until all of the numbers are non-negative.



During any step, if the lowest negative number appears at more than one corner, use the one that would be found first, if you started at the top corner and traversed in clockwise direction.

Given the original five numbers at the corners of a pentagon, output the Positive Pentagon that can be created by following this process. You may assume this process will always make a pentagon "pentastic" in at most 1000 steps.

Input

The first input line will contain only a single positive integer N, which is the number of pentagons to process. The next N lines will contain pentagon descriptions, one per line. Each pentagon description will consist of exactly 5 integers, which are in the range -999 to 999 (inclusive), and which sum up to a positive number less than 1000.

There will be exactly one space between numbers, and no leading or trailing spaces on the input lines. Positive numbers in the input will not have a leading '+' sign. The numbers are given in a clockwise order around the pentagon, starting from the top. This means that the 1^{st} and 3^{rd} numbers are neighbors of the 2^{nd} number, the 5^{th} and 2^{nd} numbers are neighbors of the 1^{st} number, and so on.

Output

For each pentagon in the input, output the message `Pentagon #p:`, where p is the pentagon number (starting from 1). Then, for each pentagon, output the Positive Pentagon that results from applying the process described above. Output the numbers for each corner using the same ordering and method used in the input, with number for the top corner first, and the others following a clockwise order. Output one space between output numbers.

Follow the format illustrated in the Example Output.

Example

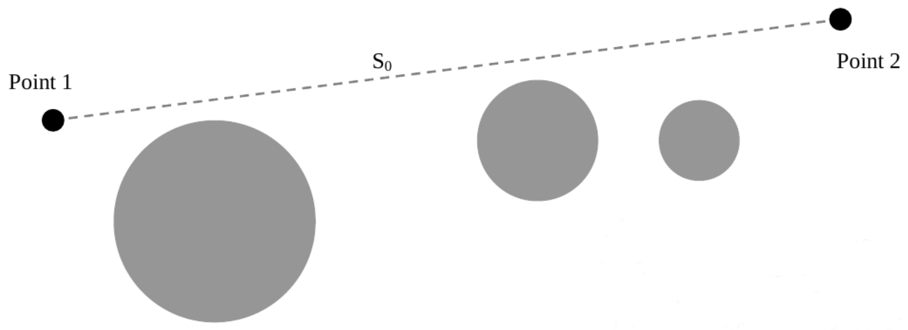
Input	Output

2
2 -1 5 7 -4
99 -1 -1 4 0
	Pentagon #1:
	1 2 2 3 1
	Pentagon #2:
	97 1 1 2 0

Problem I. Walking in the Sun

Source file name:	sunwalk.c, sunwalk.cpp, sunwalk.java, sunwalk.py
Input:	`standard`
Output:	`standard`

You have calculated all the shortest distances between points on campus, but have found that this strategy is not optimal in the summer. Some of these shortest paths are in the sun, and it's far more annoying to be sweating than to walk a few extra steps in the shade. Your goal is to recalculate shortest distances in the sun on campus, given information about where shade is located.



The original shortest path S_0 is entirely in the sun. Is it possible to spend less time in the sun between Point 1 and Point 2 by finding a path through shaded areas?

We simplify the problem by specifying all locations on campus by 2-D Cartesian coordinates, and by specifying all areas of shade as circular areas with a given center and radius. We also assume that any straight line between two Cartesian points can be walked, i.e., there are no objects blocking any possible straight line paths. Instead of calculating the shortest distance between locations on campus, your goal will be to calculate the least amount of walking that needs to be done in the sun to get between those two points.

Input

There will be multiple test cases in the input file. The first input line contains a positive integer, indicating the number of test cases. The first line of each test case, s $(0 \leq s < 50)$, contains the number of locations of shade on campus for that test case. Each of the next s lines contains the x-coordinate, y-coordinate, and radius, respectively, of a shade location, separated by spaces. The next line of each test case, q $(0 < q < 100)$, contains the number of distance queries for the test case. Each of the next q lines will contain four numbers representing two points on campus, in the form x_1 y_1 x_2 y_2, where (x_1, y_1) are the coordinates of the first point, and (x_2, y_2) are the coordinates of the second point.

All x-coordinates, y-coordinates, and radii will be real numbers between -1000 and 1000, inclusive.

Output

At the beginning of each test case, output `Campus #c:`, where c is the test case number (starting from 1) on the first line. For the subsequent q lines, begin the output with the header `Path #p:`, where p is the distance query number (starting from 1) for that case. Follow each of these headers with the statement of the form: `Shortest sun distance is D.`, where D is the desired shortest sun distance rounded to one decimal place. To clarify "rounded to one decimal place": the output for 1.74 should be 1.7, for 1.75 should be 1.8, and for 1.76 should be 1.8.

Follow the format illustrated in Example Output.

Example

Input

```
2
3
5.2 3.3 4.7
-8.8 -6.1 3.1
18.5 6.1 2.2
6
1.1 20.2 6.1 18.1
1.1 20.2 3.3 -2.5
0.4 -2.7 3.3 -2.5
6.1 18.1 -5.5 -9.2
3.3 -2.5 0.4 -2.7
1.1 20.2 0.4 -2.7
1
0.0 0.0 20.0
1
3.1 2.2 7.7 8.1
```

Output

```
Campus #1:
Path #1: Shortest sun distance is 5.4.
Path #2: Shortest sun distance is 14.1.
Path #3: Shortest sun distance is 2.9.
Path #4: Shortest sun distance is 20.6.
Path #5: Shortest sun distance is 2.9.
Path #6: Shortest sun distance is 15.7.
Campus #2:
Path #1: Shortest sun distance is 0.0.
```

Problem J. Shopping Spree

Source file name:	shop.c, shop.cpp, shop.java, shop.py
Input:	`standard`
Output:	`standard`

You've won a shopping spree with a very peculiar rule. The items you are allowed to take are in consecutive order, indexed 1 through n. You may select any subset of these items subject to the following constraint:

```
For each index k of items chosen for the subset
    at most half of the items with indexes 1 through k maybe in the subset
```

For example, if the item with index 10 is chosen for the subset, then your selected subset can contain at most half of the items with index 1 through 10. Similarly, if the item with index 2 is chosen for the subset, then your selected subset can contain at most half of the items with index 1 through 2. Note that "half" is an integer value so half of 10 and 11 are both 5. The only exception to the constraint is that if the item with index 1 is chosen for the subset, you can select 1 item and not zero (to be fair).

Given a list of the dollar values of items, $I_1, I_2, \ldots, I_n$, in the shopping spree, determine the maximum value you can obtain from the shopping spree subject to the above constraint.

Input

The first line of the input is a positive integer, n, indicating the number of shopping sprees that your program will have to analyze. Following this will be the descriptions of each shopping spree. Each shopping spree will be described on a single line. The first value on each of these lines will be a single positive integer, s ($s \leq 500$), representing the number of items for the shopping spree. The next s space-separated positive integers will be the values for the shopping spree items in dollars, in order. Each of these values will be less than or equal to 10^6.

Output

For each shopping spree, first output the heading `Spree #d:`, where d is the spree number, starting with 1. Then, print a single integer equal to the maximum value, in dollars, that can be obtained for that shopping spree. Follow the format illustrated in Example Output.

Example

Input	Output

2
5
1 2 3 4 5
3
12 2 4
	Spree #1: 9
	Spree #2: 12

Problem K. Polygon Restoration

Source file name:	polygon.c, polygon.cpp, polygon.java, polygon.py
Input:	`standard`
Output:	`standard`

A rectangular polygon is a closed figure with all vertices at points with integer coordinates in the XY-plane, and whose edges are all either horizontal or vertical. The vertices are all distinct, and no two edges intersect, except for neighboring edges intersecting at their common vertex. For the purposes of this problem, every horizontal edge will be adjacent to a vertical edge, and vice versa, so all angles are either 90 or 270 degrees.

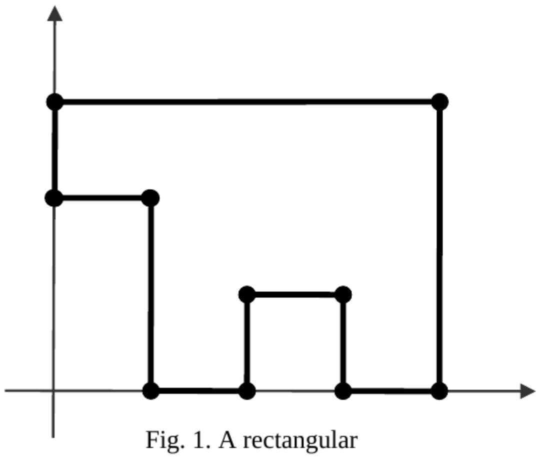


Fig. 1. A rectangular polygon in the XY-plane

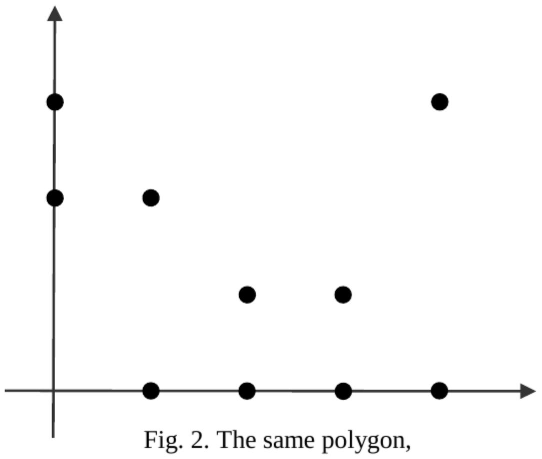


Fig. 2. The same polygon, with all of its edges erased

Dr. O is an avid collector of polygons. He takes particular pride in his collection of pencilrendered rectangular polygons from the early 19^{th} century. Unfortunately, some sneaky vandal has broken into his collection and erased all the edges, leaving only the vertices of each polygon. You have been called in to try to restore Dr. O's collection to its former glory by redrawing the edges of the rectangular polygons.

Input

There will be multiple polygons in the input. The first line of every polygon description will be an integer $N (4 \leq N \leq 50)$, the number of vertices of the polygon. Each of the next N input lines will contain two integers, giving the coordinates of a vertex of a polygon in the form "X Y" $(-10000 \leq X, Y \leq 10000)$. Note that these vertices are given in arbitrary order. All vertices will be distinct.

End of input will be indicated by a case with $N = 0$. This case should not be processed.

Output

For each test case, output a single line, formatted as: `Polygon #t:`, where t is the test case number (starting from 1), a single space, followed by the vertices of the polygon in counterclockwise order (with a single space separating vertices). Points should be referred to by their number in the order they were given in the input, the first input being vertex 1 (see Example Output for clarification). The list must start from the vertex with minimum y-coordinate. If there are multiple points with minimum y-coordinate, use the one with minimum x-coordinate. It is guaranteed that a closed rectangular polygon can always be constructed from the given data.

Example

Input

```
4
0 0
1 1
0 1
1 0
10
0 3
4 0
1 2
2 1
3 0
4 3
0 2
3 1
2 0
1 0
0
```

Output

```
Polygon #1: 1 4 2 3
Polygon #2: 10 9 4 8 5 2 6 1 7 3
```

Problem L. Super Lucky Palindromes

Source file name:	lucky.c, lucky.cpp, lucky.java, lucky.py
Input:	`standard`
Output:	`standard`

Lucky numbers are positive integers composed only of the digits '4' and '7'. For example, 47477 and 777 are lucky numbers while 457 and 1232 are not.

Super lucky numbers have the following additional properties:

- They are a lucky number themselves
- Number of digits in them is a lucky number
- The number of '4's or the number of '7's in them is a lucky number (or both counts are lucky numbers).

A palindrome is an integer that reads the same forwards and backwards. For example, 547745 and 343 are palindromes while 74 and 12345 are not. A super lucky palindrome is a positive integer that is both a super lucky number and a palindrome.

Given a number k, print the k^{th} smallest super lucky palindrome.

Input

The first input line contains a positive integer, n, indicating the quantity of numbers to check. Each of the next n lines contains a single integer, k ($1 \leq k \leq 10^{18}$).

Output

For each query, first output the heading `Query #d:`, where d is the query number, starting with 1. Then, for the value k given in the query, print the k^{th} smallest super lucky palindrome. Follow the format illustrated in Example Output.

Example

Input

```
5
1
2
3
5
100
```

Output

```
Query #1: 4444
Query #2: 7777
Query #3: 4444444
Query #4: 4747474
Query #5: 44444444744744444444444444444447447444444444
```