Problem A. Bidding

Input file: *standard input*
Output file: *standard output*

Bridge is a very complicated card game, and the bidding part is particularly difficult to master. The bidding is made even more difficult because players use different bidding conventions (meanings assigned to bids). In this problem, you are asked to write a program that suggests the first bid that should be made. The bidding conventions described below are simplified from those used by a certain person who shall remain nameless.

A bridge hand consists of 13 cards. Each card has a suit (spades, hearts, diamonds, or clubs) and a rank (A, K, Q, J, T, 9, 8, 7, 6, 5, 4, 3, 2). Here, the letter T denotes the card whose rank is 10. Before making a bid, an experienced bridge player studies the number of high card points (hcp) in the hand, as well as the distribution (the number of cards in each suit). The hcp contributed by each card is completely determined by its rank as follows:

Rank	hcp
A	4
K	3
Q	2
J	1
others	0

For example, if the hand is:

Spades:	A, 2
Hearts:	K, J, T, 9, 2
Diamonds:	3
Clubs:	K, Q, 7, 4, 3

Then this hand has 13 hcp and a distribution of 5-5-2-1 (the distribution is usually listed in non-increasing order). A balanced distribution is any one of 4-3-3-3, 4-4-3-2, and 5-3-3-2.

In bridge, an opening bid is either "pass" or consists of a level (1-7) and a trump suit. The trump suits are no trump, spades, hearts, diamonds, clubs ranked in decreasing order. Once a hand has been evaluated, the player applies the following list of (simplified) rules to determine the appropriate opening bid. In cases where multiple rules apply, the first one that applies should be used. An "x" in a distribution can be substituted with any non-negative number. Multiple "x"s in a distribution are not necessarily the same.

1. With at least 10 hcp and a y-x-x-x distribution ($y \geq 8$), bid the suit with y cards at the 4 level. This is known as a preemptive bid.
2. With 10-13 hcp and a 7-x-x-x distribution, bid the suit with 7 cards at the 3-level. This is known as a preemptive bid.
3. With 8-9 hcp and a y-x-x-x distribution ($y \geq 7$), bid the suit with y cards at the 2-level if the y-card suit is Spades or Hearts. This is known as a "weak-two" bid.
4. With 8-11 hcp and a 6-x-x-x distribution, in which Spades or Hearts is one of the 6-card suits, bid the higher rank suit at the 2 level. This is known as a "weak-two" bid.
5. With 11-15 hcp, a distribution of 4-4-4-1 or 5-4-4-0, and at least 4 spades, bid Diamonds at the 2 level. This is called the "Mini Roman Convention".

6. With 15-17 hcp and a balanced distribution, bid No Trump at the 1 level provided that at least 3 suits are "stopped". A suit is considered stopped if the suit contains at least one of the following:
   - an A;
   - a K and one other;
   - a Q and two others; or
   - a J and three others;
7. With 20-22 hcp and a balanced distribution, bid No Trump at the 2 level.
8. With at least 22 hcp, bid Clubs at the 2 level.
9. With 13-16 hcp:
   (a) If there is a 5-card or longer suit in Spades or Hearts, bid it at the 1 level. If both bids are possible, bid the longer suit. If both suits have the same length, bid the higher ranking suit.
   (b) Without a 5-card suit in Spades or Hearts, bid the longer of Diamonds or Clubs at the 1 level (whichever one has the most number of cards) . If there is a tie, bid the higher ranking suit.
10. With at least 17 hcp, bid the longest suit at the 1 level. If there is a tie, bid the lowest ranking suit. This is known as a "reverse".
11. If none of the rules above is applicable, bid Pass.

In the example above, rule 9a applies and a bid of 1 Hearts should be made.

Input

The input consists of a number of cases. The bridge hand for each case is specified on one line, with a single space separating each of the 13 cards in the hand. Each card is given as a two-character string. The first letter is the suit (S, H, D, C) and the second character is the rank (A, K, Q, J, T, 9, 8, 7, 6, 5, 4, 3, 2). The end of input is terminated by end-of-file.

Output

For each case, print the hand number (starting from 1), followed by a colon and a space, and then the suggested bid on a single line (see below for the exact format). Each bid is either "`Pass`" or a level and a suit ("`No Trump`", "`Spades`", "`Hearts`", "`Diamonds`", "`Clubs`") separated by a single space.

Example

standard input	standard output
SA S2 HK HJ HT H9 H2 D3 CK CQ C7 C4 C3 SK SQ HT H8 H4 CA CQ CT C5 DK DQ DJ D8 SA SK SQ S3 S2 HT D7 D9 CA CK CQ C7 C5	Hand #1: 1 Hearts Hand #2: 1 No Trump Hand #3: 1 Clubs

Problem B. Core Wars

Input file: *standard input*
Output file: *standard output*

Core Wars is a game in which two opposing warrior programs attempt to destroy each other in the memory of a virtual machine. They do this by overwriting each other's instructions, and the first program to execute an illegal instruction is declared the loser. Each program is written in an assembly-like language called **Redcode**, and the virtual machine which executes the two programs is known as the **Memory Array Redcode Simulator** (MARS). Your goal is to write a MARS that will read in two Redcode programs, simulate them, and print out which program was the winner.

MARS simulates a somewhat unusual environment compared to other virtual machines and processor architectures. The following list describes these differences in detail:

1. MARS simulates a machine with 8 000 memory locations and each location stores exactly one Redcode instruction. In fact, a memory location can only store instructions and cannot directly store any data. However, each instruction includes two numeric operands, and these in turn can be manipulated by other instructions for data storage. This also makes self-modifying code possible.
2. The memory locations are arranged as a continuous array with the first location having address 0 and the last having 7 999. All address calculations are performed using modulo 8 000 arithmetic. Put in another way, memory addresses wrap around so that addresses 8 000, 8 001, and 8 002 refer to same memory locations respectively as addresses 0, 1, and 2. This also works for negative numbers. For example, −7 481, −15 481, 519, and 8 519 all refer to the same memory location.
3. All arithmetic and comparison operations are performed modulo 8 000. Additions must normalize their final result to be in the range of 0 to 7 999 (inclusive) before writing that result into memory. This also implies that −124 is considered to be *greater* than 511 since after normalization, −124 becomes 7 876, and 7 876 is greater than 511.
4. The simulator maintains two separate instruction pointers (IPs) that store the address of the next instruction to be executed by the warrior programs. After loading both programs into memory, these IPs are initialized to the first instruction of each program. As the programs run, the IP is incremented by one (modulo 8 000) after each instruction is executed. If a jump/skip instruction is executed, then the IP is instead loaded with the destination address of the jump/skip and execution continues from this new address.
5. The simulator "time slices" between warriors by executing one instruction at a time, and alternating between programs after each instruction. For example, if the two programs were loaded at addresses 2 492 and 6 140, the first six instructions would be executed in this order (assuming no jump/skip instruction were executed): 2 492, 6 140, 2 493, 6 141, 2 494, 6 142.

Every instruction in MARS consists of an opcode, written as a three letter mnemonic, and two operands called the A and B fields. Each operand is a number in the range 0–7 999 (inclusive) and each can use one of three addressing modes: immediate, direct, and indirect. These modes are explained in more detail below:

- Immediate operands are written with a '`#`' sign in front, as in `#1234`. An immediate operand specifies a literal value for the instruction to operate on. For example, the first operand (i.e. the A field) of an `ADD` instruction (which performs integer addition) can be an immediate. In that case, the literal value specified by the first operand provides one of the numbers being added.
- Direct operands identify the memory locations which an instruction is to access. They are written with a '`$`' sign in front, as in `$1234`. One example would be `ADD #5 $3`. A direct operand is actually an offset relative to the current IP address. For example, if the `ADD #5 $3` instruction were stored in

memory location 4 357, it would actually be adding together a literal number five with a second value stored in the B field of location 4 360 ($4\,357 + 3$). However, if that same instruction were stored at location 132 it would be adding five to a value in the B field of location 135 ($132 + 3$).

- Indirect operands are analogous to how pointers in some programming languages work. Indirect operands are written with a '`@`' sign in front of the number, as in `ADD #5 @3`. Like before, the indirect operand is an offset relative to the current IP address, and therefore identifies a particular memory location. However, the value stored in the B field of this memory location is then used as an offset relative to that location to identify a second location. It is the B field of this second location which will actually be operated on by the instruction itself. For example, if location 4 357 contained `ADD @1 @3`, location 4 358 contained 11 in its B field, and location 4 360 contained 7 996 in its B field, then this instruction would actually be adding the values stored in locations 4 369 ($4358 + 11$) and 4 356 ($4\,360 + 7\,996$ modulo 8 000).

The list below explains what each instruction does based on its opcode. Although not all instructions use both of their operands, these must still be specified since other instructions might use these operands for data storage. **Some instructions update the B field of another instruction; this only affects the numerical value of the field, but does *not* change its addressing mode.**

- `DAT`: This instruction has two purposes. First, it can be used as a generic placeholder for arbitrary data. Second, attempting to execute this instruction terminates the simulation and the program which tried to execute it loses the match. This is the only way that a program can terminate, therefore each warrior attempts to overwrite the other one's program with `DAT` instructions. Both A and B operands must be immediate.
- `MOV`: If the A operand is immediate, the value of this operand is copied into the B field of the instruction specified by `MOV`'s B operand. If neither operand is immediate, the entire instruction (including all field values and addressing modes) at location A is copied to location B. The B operand cannot be immediate.
- `ADD`: If the A operand is immediate, its value is added to the value of the B field of the instruction specified by `ADD`'s B operand, and the final result is stored into the B field of that same instruction. If neither operand is immediate, then they both specify the locations of two instructions in memory. In this case, the A and B fields of one instruction are respectively added to the A and B fields of the second instruction, and both results are respectively written to the A and B fields of the instruction specified by the `ADD`'s B operand. The B operand cannot be immediate.
- `JMP`: Jump to the address specified by the A operand. In other words, the instruction pointer is loaded with a new address (instead of being incremented), and the next instruction executed after the `JMP` will be from the memory location specified by A. The A operand cannot be immediate. The B operand must be immediate, but is not used by this instruction.
- `JMZ`: If the B field of the instruction specified by `JMZ`'s B operand is zero, then jump to the address specified by the A operand. Neither the A nor B operand can be immediate.
- `SLT`: If A is an immediate operand, its value is compared with the value in the B field of the instruction specified by `SLT`'s B operand. If A is not immediate, the B fields of the two instructions specified by the operands are compared instead. If the first value (i.e the one specified by A) is less than the second value, then the next instruction is skipped. The B operand cannot be immediate.
- `CMP`: The entire contents of memory locations specified by A and B are checked for equality. If the two locations are equal, then the next instruction is skipped. Memory locations are considered equal to another if they both have the same opcodes and they have the same values and addressing modes in their respective operand fields. The A or B operands cannot be immediate.

Input

The input begins with a line containing a single integer n indicating the number of independant simulations to run. For each simulation the input will contain a pair of programs, designated as warrior number one and warrior number two. Each warrior program is specified using the following format:

One line with integer m ($1 \le m \le 8\,000$) indicating the number of instructions to load for this warrior. A second line containing an integer a ($0 \le a \le 7\,999$) gives the address at which to start loading the warrior's code. These two lines are then followed by m additional lines containing the warrior's instructions, with one instruction per line. If the warrior is loaded at the end of memory, the address will wrap around and the instructions continue loading from the beginning of memory.

The address ranges occupied by the two programs will not overlap. **All other memory locations which were not loaded with warrior code must be initialized to** `DAT #0 #0`. Execution always begins with warrior number one (i.e. the warrior read in first from the input file).

Output

Each simulation continues running until either warrior executes a DAT instruction or until a total of 32 000 instructions (counting both warriors) are executed. If one warrior program executes a `DAT`, the other is declared the winner; display "`Program #`x `is the winner.`", where x is either 1 or 2 and represents the number of the winning warrior. If neither program executes a `DAT` after the maximum instruction count is reached, then the programs are tied; display "`Programs are tied.`"

Example

standard input	standard output
2 3 185 ADD #4 $2 JMP $-1 #0 DAT #0 #-3 5 100 JMP $2 #0 DAT #0 #-1 ADD #5 $-1 MOV $-2 @-2 JMP $-2 #0 1 5524 MOV $0 $1 5 539 JMP $2 #0 DAT #0 #-1 ADD #5 $-1 MOV $-2 @-2 JMP $-2 #0	Program #2 is the winner. Programs are tied.

Problem C. Doors and Penguins

Input file: *standard input*
Output file: *standard output*

The organizers of the Annual Computing Meeting have invited a number of vendors to set up booths in a large exhibition hall during the meeting to showcase their latest products. As the vendors set up their booths at their assigned locations, they discovered that the organizers did not take into account an important fact — each vendor supports either the Doors operating system or the Penguin operating system, but not both. A vendor supporting one operating system does not want a booth next to one supporting another operating system.

Unfortunately the booths have already been assigned and even set up. There is no time to reassign the booths or have them moved. To make matter worse, these vendors in fact do not even want to be in the same room with vendors supporting a different operating system.

Luckily, the organizers found some portable partition screens to build a wall that can separate the two groups of vendors. They have enough material to build a wall of any length. The screens can only be used to build a straight wall. The organizers need your help to determine if it is possible to separate the two groups of vendors by a single straight wall built from the portable screens. The wall built must not touch any vendor booth (but it may be arbitrarily close to touching a booth). This will hopefully prevent one of the vendors from knocking the wall over accidentally.

Input

The input consists of a number of cases. Each case starts with 2 integers on a line separated by a single space: D and P, the number of vendors supporting the Doors and Penguins operating system, respectively ($1 \le D, P \le 500$). The next D lines specify the locations of the vendors supporting Doors. This is followed by P lines specifying the locations of the vendors supporting Penguins. The location of each vendor is specified by four positive integers: x_1, y_1, x_2, y_2. (x_1, y_1) specifies the coordinates of the southwest corner of the booth while (x_2, y_2) specifies the coordinates of the northeast corner. The coordinates satisfy $x_1 < x_2$ and $y_1 < y_2$. All booths are rectangular and have sides parallel to one of the compass directions. The coordinates of the southwest corner of the exhibition hall is $(0, 0)$ and the coordinates of the northeast corner is $(15\,000, 15\,000)$. You may assume that all vendor booths are completely inside the exhibition hall and do not touch the walls of the hall. The booths do not overlap or touch each other.

The end of input is indicated by $D = P = 0$.

Output

For each case, print the case number (starting from 1), followed by a colon and a space. Next, print the sentence:

`It is possible to separate the two groups of vendors.`

if it is possible to do so. Otherwise, print the sentence:

`It is not possible to separate the two groups of vendors.`

Print a blank line between consecutive cases.

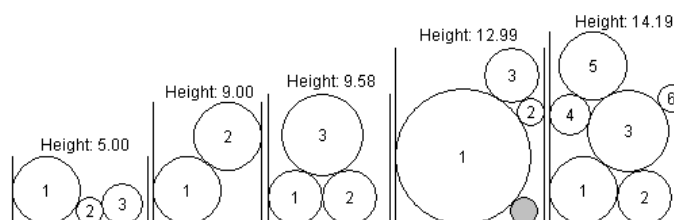
Example

standard input
3 3 10 40 20 50 50 80 60 90 30 60 40 70 30 30 40 40 50 50 60 60 10 10 20 20 2 1 10 10 20 20 40 10 50 20 25 12 35 40 0 0

standard output
Case 1: It is possible to separate the two groups of vendors. Case 2: It is not possible to separate the two groups of vendors.

Problem D. Falling Ice

Input file: *standard input*
Output file: *standard output*



Imagine disks of ice falling, one at a time, into a box, each ending up at the lowest point it can reach without overlapping or moving previous disks. Each disk then freezes into place, so it cannot be moved by later disks. Your job is to find the overall height of the final combination of disks.

So that the answer is unique, assume that any disk reaching the bottom of the box rolls as far to the left as possible. Also the data is chosen so there will be a unique lowest position for any disk that does not reach the bottom. The data is also such that there are no “perfect fits”: each disk that lands will be in contact with only two other points, on previous circles or the sides of the box. The illustrations above show white filled disks labeled with the order in which they fall into their boxes. The gray circle in the fourth illustration is not intended to be a disk that fell in. The gray disk is included to demonstrate a point: the gray disk is the same size as disk 2, so there is *space* for disk 2 on the very bottom of its box, but disk 2 cannot *reach* that position by falling from the top. It gets caught on disk 1 and the side of the box.

Input

The input consists of one or more data sets, followed by a line containing only 0 that signals the end of the input. Each data set is on a line by itself and contains a sequence of three or more blank-separated positive integers, in the format w, n, d_1, d_2, d_3, $\ldots$, d_n, where w is the width of the box, n is the number of disks, and the remaining numbers are the diameters of the disks, in the order in which they fall into the box. You can assume that $w < 100$, that $n < 10$, and that each diameter is less than w.

Output

For each data set, output a single line containing the height of the pile of disks, rounded to two places beyond the decimal point.

Example

The example data matches the illustrations above.

standard input	standard output
10 3 5 2 3 8 2 5 5 11 3 10 2 4 9 3 4 4 6 10 6 5 4 6 3 5 2 0	5.00 9.00 12.99 9.58 14.19

Problem E. Human Knot

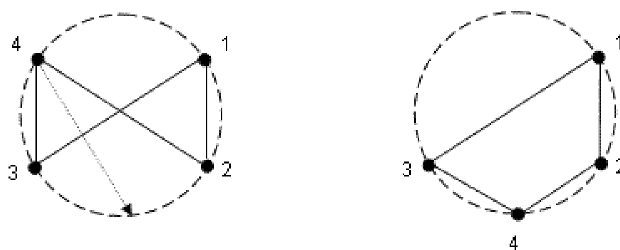
Input file: *standard input*
Output file: *standard output*

A classic ice-breaking exercise is for a group of n people to form a circle and then arbitrarily join hands with one another. This forms a "human knot" since the players' arms are most likely intertwined. The goal is then to unwind the knot to form a circle of players with no arms crossed.

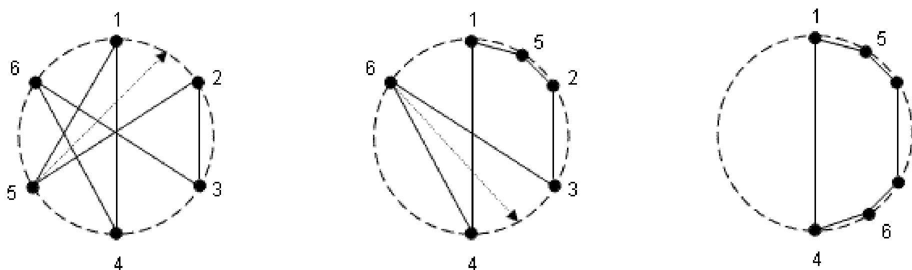
We now adapt this game to a more general and more abstract setting where the physical constraints of the problem are gone. Suppose we represent the initial knot with a 2-regular graph inscribed in a circle (i.e., we have a graph with n vertices with exactly two edges incident on each vertex). Initially, some edges may cross other edges and this is undesirable. This is the "knot" we wish to unwind.

A "move" involves moving any vertex to a new position on the circle, keeping its edges intact. Our goal is to make the fewest possible moves such that we obtain one n-sided polygon with no edge-crossings remaining.

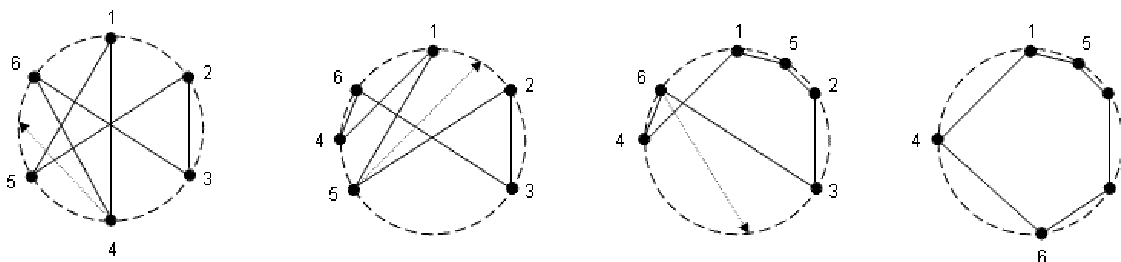
For example, here is a knot on 4 vertices inscribed in a circle, but two edges cross each other. By moving vertex 4 down to the position between 2 and 3, a graph without edge-crossings emerges. This was achieved in a single move, which is clearly optimal in this case.



When n is larger, things may not be quite as clear. Below we see a knot on 6 vertices. We might consider moving vertex 4 between 5 and 6, then vertex 5 between 1 and 2, and finally vertex 6 between 3 and 4; this unwinds the knot in 3 moves.



But clearly we can unwind the same knot in only two moves:



Input

The input consists of a number of cases. Each case starts with a line containing the integer n ($3 \leq n \leq 500$), giving the number of vertices of the graph. The vertices are labelled clockwise from 1 to n. Each of the next n lines gives a pair of neighbors, where line i ($1 \leq i \leq n$) specifies the two vertices adjacent to vertex i. The input is terminated by $n = 0$.

Output

For each case, if there is no solution, print "`Not solvable.`" on a line by itself. If there is a solution, print "`Knot solvable.`" on a line by itself, followed by the minimum number of moves required to solve the problem, on a line by itself.

Example

standard input	standard output
6 4 5 3 5 2 6 1 6 1 2 3 4 6 2 6 1 4 5 6 2 5 3 4 1 3 0	Knot solvable. 2 Knot solvable. 1

Problem F. Knots

Input file: *standard input*
Output file: *standard output*

An even number N of strands are stuck through a wall. On one side of the wall, a girl ties $N/2$ knots between disjoint pairs of strands. On the other side of the wall, the girl's groom-to-be also ties $N/2$ knots between disjoint pairs of strands. You are to find the probability that the knotted strands form one big loop (in which case the couple will be allowed to marry).

For example, suppose that $N = 4$ and you number the strands $1, 2, 3, 4$. Also suppose that the girl has created the following pairs of strands by tying knots: $\{(1, 4), (2, 3)\}$. Then the groom-to-be has two choices for tying the knots on his side: $\{(1, 2), (3, 4)\}$ or $\{(1, 3), (2, 4)\}$.

Input

The input file consists of one or more lines. Each line of the input file contains a positive even integer, less than or equal to 100. This integer represents the number of strands in the wall.

Output

For each line of input, the program will produce exactly one line of output: the probability that the knotted strands form one big loop, given the number of strands on the corresponding line of input. Print the probability to 5 decimal places.

Example

standard input	standard output
4 20	0.66667 0.28377

Problem G. Marbles in Three Baskets

Input file: *standard input*
Output file: *standard output*

Each of three baskets contains a certain number of marbles. You may move from one basket into another basket as many marbles as are already there, thus doubling the quantity in the basket that received the marbles. You must find a sequence of moves that will yield the same number of marbles in the three baskets. Moreover, you must achieve the goal in the smallest possible number of moves. Your program must also recognize the case in which there is no such sequence of moves.

Input

Each line of the input file will contain data for one instance of the problem: three positive integers, with one blank space separating adjacent integers. The three integers represent the initial numbers of marbles in the three baskets. The sum of the three integers will be at most 60.

Output

The output will begin with the initial configuration from the input. Thereafter, on successive lines, the number of marbles in the respective baskets will be printed after each move, concluding with the line in which the three numbers are identical. As stated above, the goal must be achieved in the smallest possible number of moves. (The correct output is not unique, however. There may be different sequences of moves which achieve the goal correctly in the smallest possible number of steps.) If there is no sequence of moves to achieve the goal, only the initial configuration will be printed. Each integer in the output will be right-justified in a field of width 4. Each instance of the problem will be concluded by a line of 12 equal signs.

Example

standard input	standard output

<pre>6 7 11
15 18 3
5 6 7</pre>
	<pre> 6 7 11
	   6 14 4
	  12 8 4
	   8 8 8
	============
	  15 18 3
	  12 18 6
	  12 12 12
	============
	   5 6 7
	============</pre>

Problem H. Margaritas on the River Walk

Input file: *standard input*
Output file: *standard output*

One of the more popular activities in San Antonio is to enjoy margaritas in the park along the river know as the *River Walk*. Margaritas may be purchased at many establishments along the River Walk from fancy hotels to *Joe's Taco and Margarita stand*. (The problem is not to find out how Joe got a liquor license. That involves Texas politics and thus is much too difficult for an ACM contest problem.) The prices of the margaritas vary depending on the amount and quality of the ingredients and the ambience of the establishment. You have allocated a certain amount of money to sampling different margaritas.

Given the price of a single margarita (including applicable taxes and gratuities) at each of the various establishments and the amount allocated to sampling the margaritas, find out how many different maximal combinations, choosing at most one margarita from each establishment, you can purchase. A valid combination must have a total price no more than the allocated amount and the unused amount (*allocated amount — total price*) must be less than the price of any establishment that was not selected. (Otherwise you could add that establishment to the combination.)

For example, suppose you have $25 to spend and the prices (whole dollar amounts) are:

Vendor	A	B	C	D	H	J
Price	8	9	8	7	16	5

Then possible combinations (with their prices) are: ABC(25), ABD(24), ABJ(22), ACD(23), ACJ(21), ADJ(20), AH(24), BCD(24), BCJ(22), BDJ(21), BH(25), CDJ(20), CH(24), DH(23) and HJ(21).

Thus the total number of combinations is 15.

Input

The input begins with a line containing an integer value specifying the number of datasets that follow, N, ($1 \leq N \leq 1\,000$). Each dataset starts with a line containing two integer values V and D representing the number of vendors ($1 \leq V \leq 30$) and the dollar amount to spend ($1 \leq D \leq 1\,000$) respectively. The two values will be separated by one or more spaces. The remainder of each dataset consists of one or more lines, each containing one or more integer values representing the cost of a margarita for each vendor. There will be a total of V cost values specified. The cost of a margarita is always at least one (1). Input values will be chosen so the result will fit in a 32 bit unsigned integer.

Output

For each problem instance, the output will be a single line containing the dataset number, followed by a single space and then the number of combinations for that problem instance.

Note: Some solution methods for this problem may be exponential in the number of vendors. For these methods, the time limit may be exceeded on problem instances with a large number of vendors such as the second example below.

Example

<table>
<tr><th>standard input</th><th>standard output</th></tr>
<tr><td>2
6 25
8 9 8 7 16 5
30 250
1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30</td><td>1 15
2 16509438</td></tr>
</table>

Problem I. String Equations

Input file: *standard input*
Output file: *standard output*

We all understand equations such as:

$$3 + 8 = 4 + 7$$

But what happens if we look at equations with strings instead of numbers? What would addition and equality mean?

Given two strings x and y, we define $x + y$ to be the concatenation of the two strings. We also define $x = y$ to mean that x is an anagram of y. That is, the characters in x can be permuted to form y.

You are given n distinct nonempty strings, each containing at most 10 lowercase characters. You may also assume that at most 10 distinct characters appear in all the strings. You need to determine if you can choose strings to put on both sides of an equation such that the "sums" on each side are "equal" (by our definitions above). You may use each string on either side 0 or more times, but no string may be used on both sides.

Input

The input consists of a number of cases. Each case starts with a line containing the integer n ($2 \le n \le 100$). The next n lines contain the n strings. The input is terminated with $n = 0$.

Output

For each case, print either "`yes`" or "`no`" on one line indicating whether it is possible to form an equation as described above. If it is possible, print on each of the next n lines how many times each string is used, with the strings listed in the same order as the input. On each line, print the string, followed by a space, followed by the letter "`L`", "`R`", or "`N`" indicating whether the string appears on the left side, the right side, or neither side in the equation. Finally, this is followed by a space and an integer indicating how many times the string appears in the equation. Each numeric output should fit in a 64-bit integer.

If there are multiple solutions, any solution is acceptable.

Example

standard input	standard output
2 hello world 7 i am lord voldemort tom marvolo riddle 0	no yes i L 1 am L 1 lord L 1 voldemort L 1 tom R 1 marvolo R 1 riddle R 1

Problem J. Yellow Code

Input file: *standard input*
Output file: *standard output*

nspired by Gray code, professor John Yellow has invented his own code. Unlike in Gray code, in Yellow code the adjacent words have many different bits.

More precisely, for $s = 2^n$ we call the permutation $a_1, a_2, \ldots, a_s$ of all n-bit binary words the *Yellow code* if for all $1 \le k < s$ words a_k and a_{k+1} have at least $\lfloor n/2 \rfloor$ different bits and a_1 and a_s also have at least $\lfloor n/2 \rfloor$ different bits.

Given n you have to find the n-bit Yellow code or detect that there is none.

Input

Input file contains the number n ($2 \le n \le 12$).

Output

Output 2^n n-bit binary vectors in the order they appear in some n-bit Yellow code, one on a line. If there is no n-bit Yellow code, output "`none`" on the first line of the output file.

Example

standard input	standard output
4	0000 1111 0001 1110 0010 1101 0011 1100 0101 1011 0100 1010 0110 1000 0111 1001

Problem K. Toothpick Arithmetic

Input file: *standard input*
Output file: *standard output*

A toothpick expression uses toothpicks to represent a positive integer. The expression consists of operands and operators.

Each operand consists of one or more vertical toothpicks ("|"); the value of the operand is the number of toothpicks.

The operators that can appear in an expression are addition and multiplication. The addition operator is the plus sign ("+"), which consists of one vertical and one horizontal toothpick. The multiplication operator is the letter "x", which also consists of two toothpicks. Multiplication has precedence over addition.

The expression must begin with an operand. Thereafter, operators and operands alternate. Finally, the expression must end with an operand. Given a positive integer, your program must represent it as a toothpick expression, using the smallest number of toothpicks.

Input

The input file will consist of one or more lines; each line will contain data for one instance of the problem. More specifically, each line will contain one positive integer, not exceeding 5 000.

Output

Each line of input will give rise to one line of output, consisting of: the number of toothpicks used in the expression, the expression, and the given integer from the input, formatted as shown in the sample output. The word "toothpicks" (even if the answer is 1) will be preceded by one blank space and followed by a colon and one blank space. An equal sign (but no blank spaces) will separate the expression from the given number. The expression should not contain any spaces.

If there are multiple expressions which use the smallest number of toothpicks, any such expression is acceptable.

Example

<table>
<tr><th>standard input</th><th>standard output</th></tr>
<tr><td>35
37
53</td><td>14 toothpicks: |||||||x|||||=35
17 toothpicks: ||||||x||||||+|=37
21 toothpicks: |||||x|||||x||+|||=53</td></tr>
</table>