Q1: In Penny Lane There Is A Barber Selling Photographs

Given a dollar amount between $0.01 and $9.99 you are to compute the fewest number of standard American coins needed to produce that sum. The standard coins are:

- dollar ($1.00)
- half-dollar ($0.50)
- quarter ($0.25)
- dime ($0.10)
- nickel ($0.05)
- penny ($0.01)

Sample	Results

`$1.47`	`1 dollar`
	`1 quarter`
	`2 dimes`
	`2 pennies`

You must output the number of coins in descending order of value, being careful to use plurals when a specific coin is represented more than once. Note that the plural of "penny" is "pennies".

Input

The input will be a string on a line by itself containing 5 characters of the form `$d.pq` where `d`, `p`, and `q` are digits in the range of 0-9. The value represented by this string is in the range of $0.01 and $9.99.

Output

Your output will contain a number of lines, sorted by monetary value. Each line contains a string of the form:

n unit

where *n* is an integer greater than 0 and *unit* is a standard unit of American coinage, properly pluralized when n > 1. Note that each line contains a single space between *n* and *unit*. The unit on each subsequent line of output is smaller in monetary value than the unit on the previous line. All letters in the output must be in lowercase.

Sample Input and Output

Input	Output
`$0.13`	`1 dime` `3 pennies`

`$9.87`	`9 dollars`
	`1 half-dollar`
	`1 quarter`
	`1 dime`
	`2 pennies`

`$7.00`	`7 dollars`

`$3.31`	`3 dollars`
	`1 quarter`
	`1 nickel`
	`1 penny`

`$0.01`	`1 penny`

Q2: Rinse. Lather. Repeat. Rinse. Lather. Repeat…

Given a set of $1 \le n \le 5$ nested **for** statement loops, how many times does the innermost BODY execute? In a **for** statement loop, an integer *loop variable* is incremented by 1 starting from an initial minimum value until it reaches a terminating maximum value.

Consider the two examples in the table on the right. In example **E1**, the outer `i` loop iterates 3 times while the inner `j` loop iterates 6 times, thus BODY executes 18 times. In example **E2**, the outer `k` loop iterates 3 times but the inner `j` loop iterates a variable number of times. The first time through, the `j` loop iterates 5 times, then the second time it iterates 4 times, and in the third and final time, it iterates 3 times, for a total of 12 executions of the innermost BODY.

#	Sample Code With Nested For Statements	Number of times BODY executes

E1	`for i = 1 to 3 do`
	`  for j = 2 to 7 do`
	`  BODY`
	`  end`
	`end`
		18

E2	`for k = 1 to 3 do`
	`  for j = k to 5 do`
	`  BODY`
	`  end`
	`end`
		12

The syntax of the **for** statement is defined below:

stmt := **for** *var* **=** *expr* **to** expr **do**
expr := *var* | **0** .. **9**
var := **a** .. **z**

Your program shall read the declaration of up to 5 nested **for** statement loops and output how often the innermost BODY executes. You can assume that: all variables are lowercase letters; any numeric expression is a digit in the range 0 .. 9; no *loop variable* is duplicated in any provided input; any *loop variable* referenced in an expression is properly defined in an earlier **for** statement loop (as in example **E2**); and the input is grammatically correct. As with any **for** statement loop, if the initial minimum value is already greater than the terminating maximum value, then the **for** statement loop does not execute.

Input

The first line of input will be an integer on a line by itself representing the number of nested **for** statement loops, *n*; you can assume $1 \le n \le 5$. Each of the subsequent *n* lines of input will contain the declaration of a single **for** statement. The input will be both syntactically and semantically correct.

Output

Your output will contain a single integer on a line by itself representing the number of times the innermost BODY would execute.

Sample Input and Output

Input	Output
`1` `for x = 1 to 9 do`	`9`

`2`
`for y = 2 to 9 do`
`for z = y to 8 do`
	`28`

`1` `for i = 9 to 2 do`	`0`

`3`
`for a = 1 to 9 do`
`for b = a to 7 do`
`for c = a to b do`
	`84`

Q3: A Cold Compress Cures Everything

An ASCII string represents each character using 8 full bits. In many ways this is quite wasteful for common strings composed from a mix of upper- and lower-case characters and spaces. Consider using just three bits to represent 7 high-frequency characters as shown on right. In this way, the string "noise" would require just 15-bits as "`101|011|100|110|000`". However, you must represent strings with other characters as well. For an arbitrary string, construct a sequence of *bit buffers*, whose structure is shown below; each buffer is composed of a 4-bit header, an arbitrary-shaped contents of anywhere between 3-56 bits, and an optional zero padding trailer of up to three bits that ensures each bit-buffer contains a number of bits evenly divisible by four. Each buffer represents up to 7 letters.

Encoding	Char.
000	e
001	t
010	a
011	o

Encoding	Char.
100	i
101	n
110	s
111	SPACE (' ')

	Original String	Header		Contents	Optional Zero Padding Trailer
		Type (1 bit)	Count (3 bits)		
Uncompressed	"Th"	`0`	`010`	`0101010001101000`	
Compressed	"noise"	`1`	`101`	`101011100110000`	`0`

Note that 'T' (ASCII of 84) is 01010100 in binary, as you can see in the start of the contents for the uncompressed row. These buffers are concatenated together from left-to-right. To compress the 9-character string "The noise", for example, concatenate a 20-bit uncompressed buffer (for "Th") with a 24-bit compressed buffer (for "e noise"), resulting in a 44-bit total shown below in base-16 notation.

Uncompressed Buffer for "**Th**"	Compressed buffer for "**e noise**"	Base-16 Output
`00100101010001101000`	`111100011110101110011000000`	`25468F1EB980`

Input

Your input will be a single string on a line by itself, composed of between 1 and 32 valid characters. A valid character is an uppercase letter – 'A' (ASCII of 65) to 'Z' (ASCII of 90) – or lowercase letter – 'a' (ASCII of 97) to 'z' (ASCII of 122) or a space ' ' (ASCII of 32).

Output

Your output will be a single string of base-16 characters (**A** .. **F** and **0** .. **9**) on a line by itself.

Sample Input and Output

Input	Output
`noise`	`DAE60`
`Th`	`25468`
`Programming Contest`	`2507296267729426D6DA941679E143E748C4`
`soon it is easiest`	`FCDDF08FF370B0C831`
`e`	`90`
`A`	`141`
`Thee`	`25468A00`

Q4: Triple Play

Given integer N (where $7 \le N \le 100$) you are to output in ascending order the set of all triples (X_0, X_1, X_2) such that:

- each element X_i is a perfect square (that is, $X_i = k * k$ for some integer k)
- the numbers X_0, X_1, X_2 form an arithmetic sequence with $X_1 = X_0 + h$ and $X_2 = X_1 + h$ for some integer h
- X_0, X_1, X_2 are each $\le N^2$

For example, "1 25 49" forms such a triple because each number is a perfect square, and the arithmetic sequence is formed with $h = 24$. In fact, this is the smallest such triple.

Input

The input will be an integer N on a line by itself where $7 \le N \le 100$.

Output

Your output will contain a number of lines, each of which contains three values – X_0, X_1, X_2 – separated by a single space. Each subsequent line of output is presented in sorted canonical order. That is, triple "y_0 y_1 y_2" appears after "x_0 x_1 x_2" if:

- $y_0 > x_0$
- ($y_0 = x_0$) and ($y_1 > x_1$)

Sample Input and Output

Input	Output
10	1 25 49
20	1 25 49 4 100 196 49 169 289
28	1 25 49 4 100 196 9 225 441 16 400 784 49 169 289 49 289 529
24	1 25 49 4 100 196 9 225 441 49 169 289 49 289 529

Q5: Anything You Can Do, I Can Do Better

A matrix M of integer values with *R* rows and *C* columns may contain a *saddle point*. A saddle point is a value in the matrix which is simultaneously the largest value in its column **and** the smallest value in its row. The matrix with *R*=3 rows and *C*=4 columns shown on the right has a saddle point of 3, because this value is the largest value in its column while also being the smallest value in its row. A matrix may have no saddle point, as you can verify from the following matrix:

4	3	4	7
2	1	7	2
1	-2	0	1

1	3	5	7
7	3	1	5

While it is possible that a matrix may have multiple saddle points, one can prove that the values of all such saddle points is the same, so you do not have to consider the issue for this problem.

Input

The first line of input contains an integer on a line by itself representing the number of rows, *R*. The second line of input contains an integer on a line by itself representing the number of columns, *C*. The next R lines of input each contains C integer values, separated by a single space, representing the input matrix M. Note that $1 < R \leq 5$ and $1 < C \leq 5$.

Output

If there is no saddle point in the input matrix M then output the string "NO SADDLE POINT" on a line by itself all in capital letters. Otherwise, output the integer value of the saddle point on a line by itself.

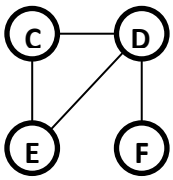
Sample Input and Output

Input	Output
2 4 1 3 5 7 7 3 1 5	NO SADDLE POINT
3 3 1 2 3 4 5 6 7 8 9	7
5 5 4 -1 -4 -3 3 3 0 1 0 2 5 -2 4 -1 -7 1 0 3 0 3 2 -1 5 -2 8	0
2 2 1 1 2 2	2

Q6: A Consummation Devoutly To Be Wished

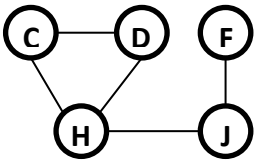
A simple, undirected graph G is defined by a set of vertices V and edges E. Each vertex in a graph is labeled by a unique capital letter and no more than one edge may exist between any two different vertices. Given two graphs, G1 and G2, you are to compute the union graph G3 = G1∪G2 such that V3 = V1∪V2 and E3 = E1∪E2. In particular, if the same letter label appears in both graphs, then the merged graph shall contain a single vertex representing the merger of the two vertices.

G1



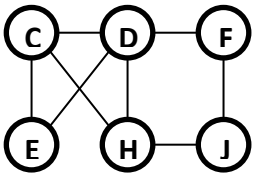
C(DE)D(CEF)E(CD)F(D)

G2



C(DH)D(CH)F(J)H(CDJ)J(FH)

G1 ∪ G2



C(DEH)D(CEFH)E(CD)F(DJ)H(CDJ)J(FH)

Note how the final merged graph has six vertices (the union of the lettered vertices in G1 and G2) and every edge from G1 and G2 also exists in G1∪G2. The string representation of a graph has each vertex appearing in ascending order, and the adjacent vertices of each vertex are listed in parentheses also in ascending order. You can verify that the strings below each graph represent the corresponding graphs.

Input

There will be two lines of input. The first line contains the string representation of graph G1 on a line by itself. The second line contains the string representation of graph G2 on a line by itself. Each graph will contain between 2 and 5 vertices and no more than 10 edges. The input will be properly formatted with each vertex appearing in ascending order and the adjacent vertices of each vertex listed in parentheses also appearing in ascending order. You can assume that every vertex has at least one edge connecting it to another vertex in the graph.

Output

You shall output the string representation of the merged graph G1∪G2 on a line by itself using the format as shown in the above example, where each vertex appears in ascending order, and the adjacent vertices of each vertex are listed in parentheses, also in ascending order.

Sample Input and Output

Input	Output
A(BC)B(A)C(A) A(XY)X(A)Y(A)	A(BCXY)B(A)C(A)X(A)Y(A)
A(B)B(A) C(D)D(C)	A(B)B(A)C(D)D(C)
A(B)B(A)C(D)D(C) A(C)C(A)	A(BC)B(A)C(AD)D(C)
A(BCD)B(ADX)C(AX)D(AB)X(BC) A(BE)B(AX)E(AFX)F(EX)X(BFE)	A(BCDE)B(ADX)C(AX)D(AB)E(AFX)F(EX)X(BCEF)

Q7: A Subset By Any Other Name

Given a set of *n* consecutive capital letters, "ABCDE…" and an integer $1 \le p \le n$, you are to output in alphabetic order all possible different subsets of size *p*. For example, given *n=4* (i.e., "ABCD") and *p=2*, there are 6 possible different subsets as shown in the adjacent table.

Different Subsets for n=4 and p=2
AB
AC
AD
BC
BD
CD

It is known that the number of different subsets of size *p* drawn from a base set of size *n* is $\frac{n!}{p!(n-p)!}$. In this example, this computes as $\frac{4!}{2!2!}$ or 6.

Input

The first line of input will be an integer on a line by itself representing the number of letters in the set, *n*. The second line of input will be an integer on a line by itself representing the size of the subsets, *p*. You can be assured that $1 \le p \le n \le 10$.

Output

The output will contain a number of lines, each of which contains a string of *p* characters in alphabetic sorted order, representing a subset drawn from the *n* letters in the original set. In addition, the entire output must appear in alphabetic sorted order as shown in the Sample Input and Output below.

Sample Input and Output

Input	Output
4 1	A B C D
5 3	ABC ABD ABE ACD ACE ADE BCD BCE BDE CDE
3 2	AB AC BC
5 5	ABCDE

Q8: When Daylight Turns to Knight

8	13	6	15
5	2	9	12
10	7	14	3
1	4	11	0

Given a 4x4 chess board, place a knight on some square and mark that square number 1. The knight makes a number of moves, never visiting the same square twice, marking each visited square with the next larger number in sequence. A valid knight's move is shaped like an "L"; it moves either two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The board shown here represents a **final valid** knight's path, starting in the lower-left corner (square marked 1) and ending in the upper right square (marked 15). Any square not visited by the knight has a mark of 0.

Your program must validate that a board containing a specific set of marked squares represents a **final valid** knight's path. A path is **valid** if the squares are marked such that moving from the square marked *i* to the square marked *i+1* can be done using a valid knight's move. A path is **final** if the knight is unable to make any valid move from the highest marked square on the board.

Input

There will be four lines of input. Each line contains four integers separated by a single space. All integers will be greater than or equal to zero and less than or equal to 16. A zero represents a square that was never visited by the knight. No number (other than 0) will be repeated in the four lines of input.

Output

Your output will be a single string on a line by itself. Output TRUE (in capital letters) if the numbered board represents a **final valid** knight's path. Output FALSE (in capital letters) if this is not the case.

Sample Input and Output

Input	Output
0 5 10 15 13 2 7 4 6 9 14 11 1 12 3 8	TRUE
11 2 13 6 8 5 10 1 3 12 7 14 0 9 4 0	TRUE
1 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0	FALSE
1 3 14 15 7 12 9 6 4 2 16 13 11 8 5 10	FALSE

Q9: A Shadow Moves Across The Moon

In high-speed finance it is often necessary to compute the "moving average" of a series of *n* values. The moving average is computed from a smaller window of p<n values as shown below. Here given n = 6 values with a window of size p = 3, the computed moving average consisting of (n – p + 1) output values is shown in the second column.

Input	Output	*Explanation*
2.0		
1.4		
2.1	1.8	*(2.0 + 1.4 + 2.1)/3 = 5.5/3 = 1.8333 truncated to 1.8*
1.9	1.8	*(1.4 + 2.1 + 1.9)/3 = 5.4/3 = 1.8 truncated to 1.8*
1.8	1.9	*(2.1 + 1.9 + 1.8)/3 = 5.8/3 = 1.93333 truncated to 1.9*
0.7	1.4	*(1.9 + 1.8 + 0.7)/3 = 4.4/3 = 1.4666 truncated to 1.4*

Each input value is of the form `u.v` (where `u` and `v` are single digits in the range 0 – 9) and you will truncate the average to one digit of precision.

Input

The first line of input will be an integer on a line by itself representing the number of values, *n*. The second line of input will be an integer on a line by itself representing the size of the window, *p*. The next *n* lines of input will be floating point numbers of the form `u.v` where `u` and `v` are single digits in the range 0-9. You can be assured that $1 < p < n < 10$.

Output

Your output will be a list of (n – p + 1) values on individual lines of the form `u.v` where `u` and `v` are single digits in the range 0 – 9.

Sample Input and Output

Input	Output
5 2 1.0 2.0 3.0 4.0 5.0	1.5 2.5 3.5 4.5
4 3 1.3 1.9 4.5 2.2	2.5 2.8
5 3 9.0 7.4 2.6 4.3 4.5	6.3 4.7 3.8