Problem 4: 24

(Problem I in Mooshak POO 2015/2016)

Submit:

- Your source code to mooshak http://mooshak.deei.fct.ualg.pt/~mooshak and
- Your report (within a zip file) to http://www.deei.fct.ualg.pt/POO/Entregas/

Up to December 14, 2015, 10H

24

As presented in

Problem E, MIUP 2007, Instituto Superior Técnico, 20 October 2007

The other day, I came across a game they play in secondary schools to develop the students mathematical capabilities. It's called the ``twenty-four game''. I understood it is played by four students sitting around a square table. There is a deck of cards that has been shuffled and is facing down at the center of the table. In each of the cards there are four non-negative integer numbers. The first student takes the top card, reads the hidden numbers, and the challenge is to devise an arithmetic expression involving those numbers, all of them, each one only once, and the four operators ``+'', ``-'', ``*'', and ``/'', whose value is 24. According to the description I got, the time the student takes to solve the riddle is recorded. Then the second student does the same, and so on, repeatedly, adding up the times for each student. After a given number of rounds, the winner is the student whose accumulated time is least.

Let us use this problem to develop our own programming capabilities (and the mathematical capabilities of our computer, while we are at it) The task is to write a program that given a list of four numbers computes all the arithmetic expressions whose value is another given number, using each of the elements of the list exactly once. Note that we are generalizing the problem, since we want to deal with any result, not necessarily 24, and we want all the expressions, not just one of them. We also clarify by writing ``using each of the elements of the list exactly once''. This means that if a number appears n times in the list, it must appear n times in the expression.

The problem is to be solved within the set of the non-negative integers. This means that all the given numbers are non-negative integers and all the intermediate results must be non-negative integers too. It implies that we cannot subtract 5 from 3, for example. Likewise, we can only divide x by y if x is a multiple of y.

To make the solution unambiguous and make the order of the operations explicit, the expressions must be completely parenthesized. For example, if we want to write the sub-expression that represents the product of x and y we should write ``(x*y)". In the place of x and y we can have any expressions, either simple numbers, for example ``7+12", or other expressions, for example ``((15-9)*(2+8))".

Input

The input file contains two lines. The first line has four numbers, all non-negative integers, less than 100, representing the numbers on the card. The second line contains one non-negative integer, representing the value of the expression.

Output

The output file contains one line for each expression. Each expression is to be written without spaces in fully parenthesized infix form and the sequence of expressions must come in lexicographical order of the corresponding string. Note that even if the numbers in the card can be repeated, there must be no duplicate expressions in the output file.

If the problem is impossible, meaning no expression exists that satisfies the requirements, the program must write a single line containing the message ``void" (without the quotes).

Sample Input 1

```
12 5 10 8
34
```

Sample Output 1

```
((10-8)*(12+5))
((10-8)*(5+12))
((12+5)*(10-8))
((5+12)*(10-8))
```

Sample Input 2

```
12 16 22 12
150
```

Sample Output 2

```
(((12*12)+22)-16)
(((12*12)-16)+22)
((12*12)+(22-16))
((22+(12*12))-16)
((22-16)+(12*12))
(22+((12*12)-16))
```

Sample Input 3

```
1 1 1 1
4
```

Sample Output 3

```
(((1+1)+1)+1)
((1+(1+1))+1)
((1+1)*(1+1))
((1+1)+(1+1))
(1+((1+1)+1))
(1+(1+(1+1)))
```

Sample Input 4

```
12 6 5 7
34
```

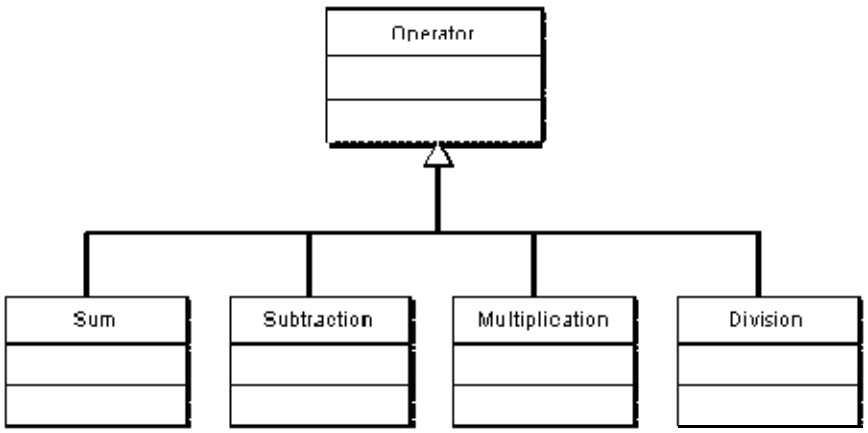
Sample Output 4

```
void
```

ADDITIONAL REQUIREMENTS

Based on the principles and techniques of object-oriented programming develop a program able to solve instances of this problem. Specially, follow the Command Design Pattern [1] and adopt the following class hierarchy which models the four required arithmetic operations.

[1] See http://en.wikipedia.org/wiki/Command_pattern and



Also take full advantage from iterators and Exception Handling. Use test-driven development.

Document in your report the unit tests developed, as well as all the design options taken.

NB: Any other approach to this problem, independently of its merit, will be marked with 0 (zero) values.