Exercises: Methods, Debugging and Troubleshooting Code

Problems for exercises and homework for the "Programming Fundamentals" course @ SoftUni.

You can check your solutions here: https://judge.softuni.bg/Contests/305/Methods-and-Debugging-Excercises.

1. Hello, Name!

Write a **method** that receives a name as **parameter** and prints on the console. "Hello, <name>!"

Examples

Input	Output
Peter	Hello, Peter!

2. Max Method

Create a method **`GetMax(int a, int b)`**, that returns the **largest** of two numbers. Write a program that reads **three numbers** from the console and **prints** the **biggest** of them. Use the **`GetMax(…)`** method you just created.

Examples

Input	Output
1 2 3	3

Input	Output
-100 -101 -102	-100

3. English Name of the Last Digit

Write a **method** that returns the **English name** of the last digit of a given number. Write a program that reads an integer and prints the returned value from this method.

Examples

Input	Output
1024	four

Input	Output
512	two

4. Numbers in Reversed Order

Write a method that **prints the digits** of a given decimal number in a **reversed order**.

Examples

Input	Output
256	652

Input	Output
1.12	21.1

5. Fibonacci Numbers

Define a method **`Fib(n)`** that calculates the n^{th} Fibonacci number. Examples:

n	Fib(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13
11	144
25	121393

6. Prime Checker

Write a Boolean method **`IsPrime(n)`** that check whether a given integer number **n** is prime. Examples:

n	IsPrime(n)
0	false
1	false
2	true
3	true
4	false
5	true
323	false
337	true
6737626471	true
117342557809	false

7. * Primes in Given Range

Write a method that calculates **all prime numbers in given range** and returns them as list of integers:

```
static List<int> FindPrimesInRange(startNum, endNum)
{
    …
}
```

Write a method to **print a list of integers**. Write a program that enters two integer numbers (each at a separate line) and prints all primes in their range, separated by a comma.

Examples

Start and End Number	Output
0 10	2, 3, 5, 7
5	5, 7, 11

11	
100 200	101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199
250 950	251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947
100 50	*(empty list)*

8. Center Point

You are given the coordinates of two points on a Cartesian coordinate system - X1, Y1, X2 and Y2. **Create a method** that prints the point that is closest to the center of the coordinate system (0, 0) in the format (X, Y). If the points are on a same distance from the center, print only the first one.

Examples

Input	Output
2 4 -1 2	(-1, 2)

9. Longer Line

You are given the coordinates of four points in the 2D plane. The first and the second pair of points form two different lines. Print the longer line in format "(X1, Y1)(X2, Y2)" starting with the point that is closer to the center of the coordinate system (0, 0) (You can reuse the method that you wrote for the previous problem). If the lines are of equal length, print only the first one.

Examples

Input	Output
2 4 -1 2 -5 -5 4 -3	(4, -3)(-5, -5)

10. Cube Properties

Write a program that can calculate the length of the face diagonals, space diagonals, volume and surface area of a **cube** (http://www.mathopenref.com/cube.html) by a given side. On the first line you will get the side of the cube. On the second line is given the parameter (**face**, **space**, **volume** or **area**).

Output should be rounded to the second digit after the decimal point:

Examples

Input	Output
5 face	7.07
5 volume	125.00

11. Geometry Calculator

Write a program that can **calculate the area** of **four different geometry figures** - triangle, square, rectangle and circle.

On the first line you will get the **figure type**. Next you will get parameters for the chosen figure, **each on a different line**:

- Triangle - side and height
- Square - side
- Rectangle - width and height
- Circle - radius

The output should be rounded to the second digit after the decimal point:

Examples

Input	Output
triangle 3 6	9.00
rectangle 4 5	20.00

12. Master Numbers

A master number is an integer that holds the following properties:

- Is **symmetric** (palindrome), e.g. 5, 77, 282, 14341, 9553559.
- Its **sum of digits is divisible by 7**, e.g. 77, 313, 464, 5225, 37173.
- Holds at least **one even digit**, e.g. 232, 707, 6886, 87578.

Write a program to **print all master numbers** in the range [1…**n**].

Examples

Input	Output
600	232 383 464 545

Input	Output
5000	232 383 464 545 626 696 707 858 1661 2552 3443 4334

Hints

1. Write 3 utility methods:
   - **`IsPalindrome(int num)`**
   - **`SumOfDigits(int num)`**
   - **`ContainsEvenDigit(int num)`**
2. Loop through all numbers in range [1…**n**] and check every number with the helper methods.

13. * Factorial

Write a program that calculates and prints the **`n!`** for any **n** in the range [1…1000].

Examples

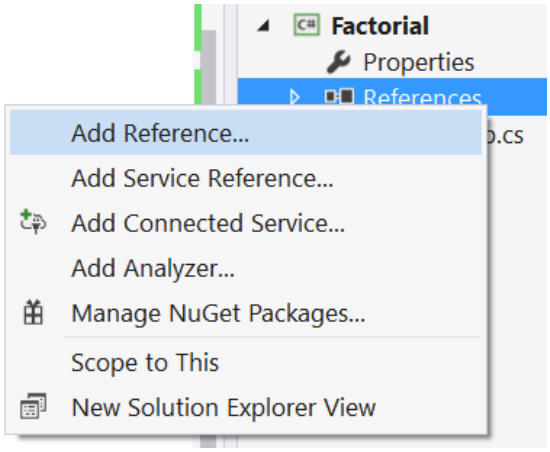
Input	Output
5	120

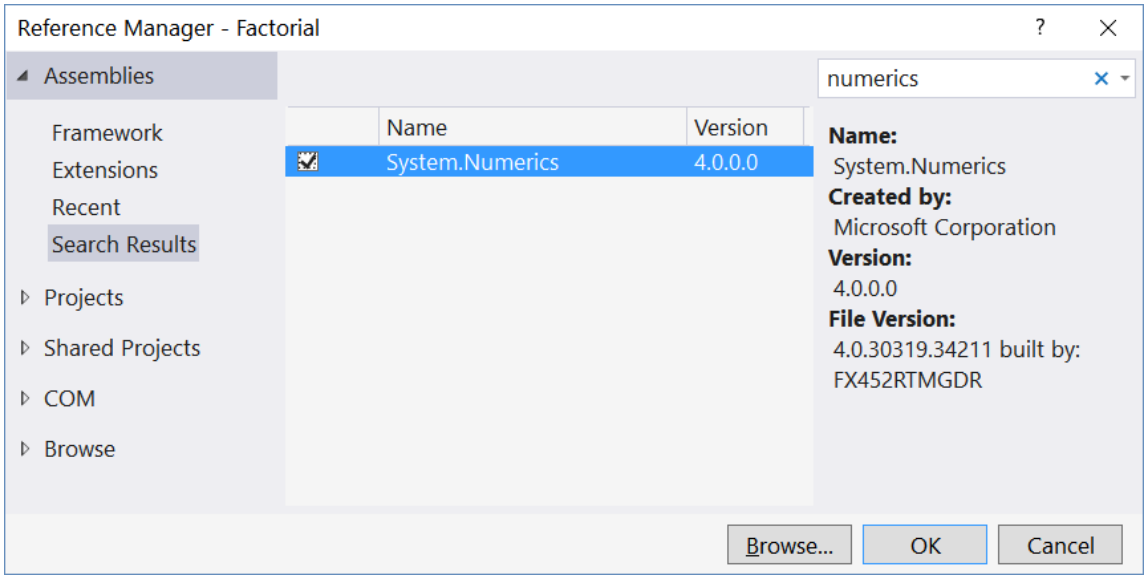
Input	Output
100	93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

Hints

Use the class **`BigInteger`** from the built-in .NET library **`System.Numerics.dll`**.

1. First add reference to **`System.Numerics.dll`**.





2. Import the namespace "**System.Numerics**":

```
using System.Numerics;
```

3. Use the type **BigInteger** instead of **long** or **decimal** to keep the factorial value:

```
BigInteger factorial = 1;
for (int i = 1; i <= n; i++)
    // TODO
```

14. Factorial Trailing Zeroes

Create a program that counts the trailing zeroes of the factorial of a given number.

Examples

Input	Output	Comments
5	1	5! = 120 -> One trailing zero
100	24	100! = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000 -> 24 trailing zeroes

Hints

1. You may use your solution from the previous problem. Add additional method that counts and returns the number of zeroes a number has.

15. ** Debugging Exercise: Substring

The goal of this exercise is to practice **debugging techniques** in scenarios where a piece of code does not work correctly. Your task is to **pinpoint the bug** and **fix it** (without rewriting the entire code). Test your fixed solution in the judge system:

You can download the broken solution from here.

Problem Description

You are given a **text** and a number **count**. Your program should search through the text for the letter '**p**' (ASCII code **112**) and print '**p**' along with **count** letters to its right.

For example, we are given the **text** "**phahah put**" and **count** = **3**. We find a match of '**p**' in the first letter so we print it and the 3 letters to its right. The result is "**phah**". We continue and find another match of '**p**', but there aren't 3 letters to its right, so we print only "**put**".

Each match should be printed on a separate line. If there are no matches of '**p**' in the text, we print "**no**".

Input

- The first line holds the **text** to be processed (string).
- The second line holds the **number count**.

Output

For each match, print the **matched substring** at separate line. Print "**no**" if there are no matches.

Constraints

- The number **count** will be in the range [0 ... 100].

Tests

Input	Program Output	Expected Output
phahah put 3	no	phah put
No match 5	no	no
preparation 4	no	prepa
preposition 0	no	P p

16. ** Debugging Exercise: Instruction Set

The goal of this exercise is to practice **debugging techniques** in scenarios where a piece of code does not work correctly. Your task is to **pinpoint the bug** and **fix it** (without rewriting the entire code).

You can download the broken solution from here.

Problem Description

Write an **instruction interpreter** that executes an arbitrary number of **instructions**. The program should **parse the instructions**, **execute** them and **print the result**. The following instruction set should be supported:

- **INC <operand1>** – increments the operand by 1
- **DEC <operand1>** – decrements the operand by 1
- **ADD <operand1> <operand2>** – performs addition on the two operands
- **MLA <operand1> <operand2>** – performs multiplication on the two operands
- **END** – end of input

Output

The result of each instruction should be printed on a separate line on the console.

Constraints

- The operands will be valid integers in the range [−2 147 483 648 … 2 147 483 647].

Tests

Input	Program Output (Wrong)	Expected Output (Correct)
INC 0 END	0 0 … (infinite)	1
ADD 1323134 421315521 END	422638655 422638655 … (infinite)	422638655
DEC 57314183 END	57314183 57314183 … (infinite)	57314182
MLA 252621 324532 END	379219748 379219748 … (infinite)	81983598372

17. ** Debugging Exercise: Be Positive

The goal of this exercise is to practice **debugging techniques** in scenarios where a piece of code does not work correctly. Your task is to **pinpoint the bug** and **fix it** (without rewriting the entire code). Test your fixed solution in the judge system:

You can download the broken solution from here.

Problem Description

A program is designed to take some **sequences of numbers** from the console, to **process them** as described below and **print** each obtained sequence.

Input

- On the first line of input you are given a **count N – the number of sequences**.
- On each of **the next N lines** you will receive some **numbers surrounded by whitespaces**.

Processing Logic

You need to check each number, if it's **positive** – print it on the console; if it's **negative**, add to its value the value of the next number and only **print the result if it's not negative**. You only perform the addition once, e.g. if you have the sequence: -3, 1, 3, the algorithm is as follows:

- -3 is negative => add to it the next number (1) => -3 + 1 = -2 still negative => do not print anything (and don't keep adding numbers, you stop here).
- The next number we consider is 3 which is positive => print it.

If no numbers can be obtained in this manner for the given sequence, print **"(empty)"**.

Example

Input	Expected Output	Comments
3 3 -4 5 2 123 -1 -1 3 4 -2 1	3 1 2 123 3 4 (empty)	(3) (-4 + 5 = 1 > 0) (2) (123) (-1 + (-1) < 0) (3) (4) (-2 + 1 < 0)

Output

Print on the console **each modified sequence on a separate line.**

Constraints

- The **number N** will be an integer in the range [1 … 15].
- The **numbers in the sequences** will be integers in the range [-1000 … 1000].
- The **count of numbers in each sequence** will be in the range [1 … 20].
- There may be **whitespaces anywhere around the numbers** in a given sequence

Tests

Input	Program Output (Wrong)	Expected Output
3 3 -4 5 2 123 -1 -1 3 4 -2 1	Exception…	3 1 2 123 3 4 (empty)
1 0 -2 2 -2 3	Exception…	0 0 1

18. ** Debugging Exercise: Sequence of Commands

The goal of this exercise is to practice **debugging techniques** in scenarios where a piece of code does not work correctly. Your task is to **pinpoint the bug** and **fix it** (without rewriting the entire code). Test your fixed solution in the judge system:

You can download the broken solution from here.

Problem Description

You are given a program that reads a **n numbers** and a **sequence of commands** to be executed over these numbers.

Input

- The first line holds an **integer n** – the **count** of numbers.
- The second line holds **n numbers** – integers separated by space.
- Each of the next few lines hold **commands** in format: **"[action] [i-th element] [value]"**.
- The commands sequence end with a command **"stop"**.

Commands

Commands are given in format **"[action] [i-th element] [value]"**. Elements are indexed from **1** to **n**.

The **action** can be **"multiply"**, **"add"**, **"subtract"**, **"rshift"** or **"lshift"**.

- The actions **"multiply", "add"** and **"subtract"** have parameters. The first parameter is the **index** of the element that needs to be changed (in range [**1**...**n**]). The second parameter is the **value** with which we manipulate the element.
- The command **"lshift"** moves the first element last. E.g. "**lshift**" over {1, 2, 3} will produce {2, 3, 1}.
- The command **"rshift"** moves the last element first. E.g. "**rshift**" over {1, 2, 3} will produce {3, 1, 2}.

Output

Print the values of the **n elements** after the execution of each command (except the last "**stop**" command).

Constraints

- The **number n** will be an integer in the range [1 … 15].
- Each **element of the array** will be an integer in the range [0 … 2^{63}-1].
- The **number `i`** and the **number of commands** will be integers in the range [1 … 10].
- The **number value** will be an integer in the range [-100 … 100]. If the command is "**rshift**" or "**lshift**" there are no parameters.

Tests

Input	Program Output (Wrong)	Expected Output
5 3 0 9 333 11 add 2 2 subtract 1 1 multiply 3 3 rshift stop	3 0 9 333 11 3 0 9 333 11	3 2 9 333 11 2 2 9 333 11 2 2 27 333 11 11 2 2 27 333