# Homework: Operators Expressions and Statements

This document defines homework assignments from the [“C# Basics“ Course @ Software University](http://softuni.bg/courses/csharp-basics/). Please submit as homework a single zip / rar / 7z archive holding the solutions (source code only) of all below described problems.

## Odd or Even Integers

Write an expression that checks if given integer is **odd or even**. Examples:

|  |  |
| --- | --- |
| **n** | **Odd?** |
| 3 | true |
| 2 | false |
| -2 | false |
| -1 | true |
| 0 | false |

## Gravitation on the Moon

The gravitational field of the Moon is approximately 17% of that on the Earth. Write a program that calculates the **weight of a man on the moon** by a given weight on the Earth. Examples:

|  |  |
| --- | --- |
| **weight** | **weight on the Moon** |
| 86 | 14.62 |
| 74.6 | 12.682 |
| 53.7 | 9.129 |

## Divide by 7 and 5

Write a Boolean expression that checks for given integer if it can be **divided** (without remainder) **by 7 and 5 in the same time**. Examples:

|  |  |
| --- | --- |
| **n** | **Divided by 7 and 5?** |
| 3 | false |
| 0 | false |
| 5 | false |
| 7 | false |
| 35 | true |
| 140 | true |

## Rectangles

Write an expression that calculates **rectangle’s perimeter** and **area** by given **width** and **height**. Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **width** | **height** | **perimeter** | **area** |
| 3 | 4 | 14 | 12 |
| 2.5 | 3 | 11 | 7.5 |
| 5 | 5 | 20 | 25 |

## Third Digit is 7?

Write **an expression** that checks for given integer **if its third digit** from right-to-left **is 7**. Examples:

|  |  |
| --- | --- |
| **n** | **Third digit 7?** |
| 5 | false |
| **7**01 | true |
| 9**7**03 | true |
| **8**77 | false |
| 777**8**77 | false |
| 9999**7**99 | true |

## Four-Digit Number

Write a program that takes as input a **four-digit number** in format **abcd** (e.g. 2011) and performs the following:

* Calculates the sum of the digits (in our example 2+0+1+1 = 4).
* Prints on the console the number in reversed order: dcba (in our example 1102).
* Puts the last digit in the first position: dabc (in our example 1201).
* Exchanges the second and the third digits: acbd (in our example 2101).

The number has always exactly **4 digits** and cannot start with 0. Examples:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n** | **sum of digits** | **reversed** | **last digit in front** | **second and third digits exchanged** |
| 2011 | 4 | 1102 | 1201 | 2101 |
| 3333 | 12 | 3333 | 3333 | 3333 |
| 9876 | 30 | 6789 | 6987 | 9786 |

## Point in a Circle

Write **an expression** that checks if given point (**x**, **y**) is inside a **circle K**({**0**, **0**}, **2**). Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **x** | **y** | **inside** |  |
| 0 | 1 | true |
| -2 | 0 | true |
| -1 | 2 | false |
| 1.5 | -1 | true |
| -1.5 | -1.5 | false |
| 100 | -30 | false |
| 0 | 0 | true |
| 0.2 | -0.8 | true |
| 0.9 | -1.93 | false |
| 1 | 1.655 | true |

## Prime Number Check

Write an **expression** that checks if given positive integer number **n** (**n** ≤ 100) is [**prime**](https://en.wikipedia.org/wiki/Prime_number) (i.e. it is divisible without remainder only to itself and 1). Examples:

|  |  |
| --- | --- |
| **n** | **Prime?** |
| 1 | false |
| 2 | true |
| 3 | true |
| 4 | false |
| 9 | false |
| 97 | true |
| 51 | false |
| -3 | false |
| 0 | false |

## Trapezoids

Write an expression that calculates **trapezoid's area** by given sides **a** and **b** and height **h**. Examples:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a** | **b** | **h** | **area** |  |
| 5 | 7 | 12 | 72 |
| 2 | 1 | 33 | 49.5 |
| 8.5 | 4.3 | 2.7 | 17.28 |
| 100 | 200 | 300 | 45000 |
| 0.222 | 0.333 | 0.555 | 0.1540125 |

## Point Inside a Circle & Outside of a Rectangle

Write an expression that checks for given point (x, y) if it is **within the circle K**({1, 1}, 1.5) and **out of the rectangle R**(top=**1**, left=**-1**, width=**6**, height=**2**). Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **x** | **y** | **inside K & outside of R** |  |
| 1 | 2 | yes |
| 2.5 | 2 | no |
| 0 | 1 | no |
| 2.5 | 1 | no |
| 2 | 0 | no |
| 4 | 0 | no |
| 2.5 | 1.5 | no |
| 2 | 1.5 | yes |
| 1 | 2.5 | yes |
| -100 | -100 | no |

## Bitwise: Extract Bit #3

Using bitwise operators, write an **expression** for finding the value of the bit #**3** of a given unsigned integer. The bits are counted from right to left, starting from bit #0. The result of the expression should be either **1 or 0**. Examples:

|  |  |  |
| --- | --- | --- |
| **n** | **binary representation** | **bit #3** |
| 5 | 00000000 0000**0**101 | 0 |
| 0 | 00000000 0000**0**000 | 0 |
| 15 | 00000000 0000**1**111 | 1 |
| 5343 | 00010100 1101**1**111 | 1 |
| 62241 | 11110011 0010**0**001 | 0 |

## Extract Bit from Integer

Write an expression that extracts from given integer **n** the value of given **bit at index** **p**. Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **binary representation** | **p** | **bit @ p** |
| 5 | 00000000 00000**1**01 | 2 | 1 |
| 0 | 000000**0**0 00000000 | 9 | 0 |
| 15 | 00000000 000011**1**1 | 1 | 1 |
| 5343 | 00010100 **1**1011111 | 7 | 1 |
| 62241 | 1111**0**011 00100001 | 11 | 0 |

## Check a Bit at Given Position

Write a **Boolean expression** that returns if the **bit at position p** (counting from **0**, starting from the right) in given integer number **n** has value of **1**. Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **binary representation of n** | **p** | **bit @ p == 1** |
| 5 | 00000000 00000**1**01 | 2 | true |
| 0 | 000000**0**0 00000000 | 9 | false |
| 15 | 00000000 000011**1**1 | 1 | true |
| 5343 | 00010100 **1**1011111 | 7 | true |
| 62241 | 1111**0**011 00100001 | 11 | false |

## Modify a Bit at Given Position

We are given an integer number **n**, a bit value **v** (v=0 or 1) and a position **p**. Write a **sequence of operators** (a few lines of C# code) that modifies **n** to hold the value **v** at the position **p** from the binary representation of **n** while preserving all other bits in **n**. Examples:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **binary representation of n** | **p** | **v** | **binary result** | **result** |
| 5 | 00000000 00000**1**01 | 2 | 0 | 00000000 00000**0**01 | 1 |
| 0 | 000000**0**0 00000000 | 9 | 1 | 000000**1**0 00000000 | 512 |
| 15 | 00000000 000011**1**1 | 1 | 1 | 00000000 000011**1**1 | 15 |
| 5343 | 00010100 **1**1011111 | 7 | 0 | 00010100 **0**1011111 | 5215 |
| 62241 | 1111**0**011 00100001 | 11 | 0 | 1111**0**011 00100001 | 62241 |

## \* Bits Exchange

Write a program that **exchanges bits** **3**, **4** and **5** with bits **24**, **25** and **26** of **given 32-bit unsigned integer**. Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **binary representation of n** | **binary result** | **result** |
| 1140867093 | 01000**100** 00000000 01000000 00**010**101 | 01000**010** 00000000 01000000 00**100**101 | 1107312677 |
| 255406592 | 00001**111** 00111001 00110010 00**000**000 | 00001**000** 00111001 00110010 00**111**000 | 137966136 |
| 4294901775 | 11111**111** 11111111 00000000 00**001**111 | 11111**001** 11111111 00000000 00**111**111 | 4194238527 |
| 5351 | 00000**000** 00000000 00010100 11**100**111 | 00000**100** 00000000 00010100 11**000**111 | 67114183 |
| 2369124121 | 10001**101** 00110101 11110111 00**011**001 | 10001**011** 00110101 11110111 00**101**001 | 2335569705 |

## \*\* Bit Exchange (Advanced)

Write a program that **exchanges bits** **{p, p+1, …, p+k-1}** with bits **{q, q+1, …, q+k-1}** of a given 32-bit unsigned integer. The first and the second sequence of bits may **not overlap**. Examples:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **p** | **q** | **k** | **binary representation of n** | **binary result** | **result** |
| 1140867093 | 3 | 24 | 3 | 01000**100** 00000000 01000000 00**010**101 | 01000**010** 00000000 01000000 00**100**101 | 1107312677 |
| 4294901775 | 24 | 3 | 3 | 11111**111** 11111111 00000000 00**001**111 | 11111**001** 11111111 00000000 00**111**111 | 4194238527 |
| 2369124121 | 2 | 22 | 10 | **10001101 00**110101 1111**0111 000110**01 | **01110001 10**110101 1111**1000 110100**01 | 1907751121 |
| 987654321 | 2 | 8 | 11 | - | - | overlapping |
| 123456789 | 26 | 0 | 7 | - | - | out of range |
| 333333333 | -1 | 0 | 33 | - | - | out of range |

# Exam problems.\*\*

All of the problems below are given from the previous C# Basics exam (26-Apr-2015-Morning). **You are not obligated** to submit any of them in your homework. We highly recommend you to try solving some or all of them so you can be well prepared for the upcoming exam. You need to learn how to use conditional statements, loops, arrays and other things (learn in internet how or read those chapters in the book “[Fundamentals of computer programming with C#](http://www.introprogramming.info/intro-csharp-book/read-online/)”). If you still find those problems too hard for solving it’s very useful to **check** and **understand** the solutions. You can download all solutions and tests for this variant [here](http://svn.softuni.org/admin/svn/csharp-basics/Feb-2015/Programming-Basics-Exam-26-Apr-2015-Morning.zip) or check all [previous exams](https://softuni.bg/courses/programming-basics) (scroll down to the bottom of the page). You can also test your solutions in our automated [judge system](https://judge.softuni.bg/Contests/81/Programming-Basics-Exam-26-April-2015-Morning) to see if you pass all tests.

## \*\* Book Problem

Stefan is a keen reader. He wants to read a programming book and wants to know exactly when he will finish it.

**Every day** in a given month **he reads up some pages**. Some days he goes campingand on these dayshe **doesn’t read**.

You will be given **the book’s page count, the number of camping days in a month** and **number of pages which Stefan reads in a normal day,** each on a separate line. Assume **each month has 30 days** and **each year has 12 months**. Calculate how many years and months Stefan will need in order to read the book and print the result on the console in format **“X years Y months”. If Stefan never reads the book, print “never”.**

**Note** that if, for example, Stefan needs 3.1 months, you need to **round that up** – so you have to print “0 years 4 months”.

### Input

The input will be read from the console. It consists of exactly **three lines**:

* On the **first line** you will be given an integer – **the number of pages of the book**.
* On the **second line** you will be given the **number of camping days in a month**.
* On the **third line** you will be given the **number of pages which Stefan reads every normal day**.

The input will always be valid and in the format described, there is no need to check it explicitly.

### Output

The output should be printed on the console.

* On the only output line **print the number of years and months** Stefan will need to read the book in **format “X years Y months”**.
* If Stefan cannot read the book, you should print **“never”**.

### Constraints

* The number of **pages** of the book will be an integer in the range [1 … 2 000 000 000].
* The number of **camping days** will be an integer in the range [0 … 30].
* The **number of pages Stefan reads** in a normal day will be an integer in the range [0 … 100].
* Allowed working time: 0.1 seconds. Allowed memory: 16 MB.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 250000  5  10 | 83 years 4 months | There are 30 days in a month => 5 camping days and 25 normal days. On a normal day he reads 10 => 25\*10 = 250 pages. On a camping day he doesn’t read.  250000/250 = 1000 – he needs exactly 1000 months. This is 83 years and 4 months. |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 25  30  100 | never |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 24  0  1 | 0 years 1 months |

## \*\* Chessboard Game

Goshko is a keen chess player. One day he was bored with his work and decided to take a break and create a game using the chessboard. He takes a string, e.g. "Software University\_2345", converts its symbols to numbers through **their ASCII codes** and fills a chessboard with them. He takes the values of **capital and small letters and digits** only. The value of **any other symbol** is **zero**. He fills the board’s squares with the numbers, from left to right and from top to bottom (see the example below). **The size of the chessboard is n\*n (e.g. n = 5) and it always starts with a black square**. **N** will always be an **odd number**.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| S | o | f | t | w |  | 83 | 111 | 102 | 116 | 119 |
| a | r | e |  | U |  | 97 | 114 | 101 | 0 | 85 |
| n | i | v | e | r |  | 110 | 105 | 118 | 101 | 114 |
| s | i | t | y | \_ |  | 115 | 105 | 116 | 121 | 0 |
| 2 | 3 | 4 | 5 |  |  | 50 | 51 | 52 | 53 | 0 |

Let’s assume that there are two competing teams: the **black team** and the **white team**. Every team’s **score** is the **sum of the values in its squares**. However if a square contains a **capital letter** its value should be **given to the opposing team**. In the example above the scores are calculated as follows:

**White Team** Score = 83 'S' + 111 'o' + 116 't' + 97 'a' + 101 'e' + 105 'i' + 101 'e' + 115 's' + 116 't' + 51 '3' + 53 '5' = 1049  
**Black Team** Score = 102 'f' + 119 'w' + 114 'r' + 85 'U' + 110 'n' + 118 'v' + 114 'r' + 105 'i' + 121 'y' + 50 '2' + 52 '4' = 1090.

### Input

The input data should be read from the console.

* The **first line** holds the **size n** of the chessboard.
* The **second line** holds the input string.

The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

The output should be printed on the console.

* The first output line holds the **winning team** in format: “The winner is: **{name}** team”.
* The second line holds the difference between the scores of the winning and the losing team.
* In case the score is **equal**, print “Equal result: **{points}**”. Do not print the difference in this case!

### Constraints

* The number **n** will be an **odd** **integer** in the range [1 … 9].
* Allowed working time for your program: 0.1 seconds. Allowed memory: 16 MB.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  Software University\_2345 | The winner is: black team  41 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  aa | Equal result: 97 |

## \*\* King of Thieves

Once upon a time there was a kingdom and everyone in the kingdom was a thief. Izzy wanted to become the King of Thieves and so started stealing only **perfect gems** from other thieves. Help Izzy by showing him what a perfect gem with given parameters should look like.

### Input

The input data should be read from the console.

* The first line will hold the **size** of the gem – **n**.
* The second line will hold the **type** of the gem – a symbol: e.g. ‘**\***’.

The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

The output should be printed on the console. It should consist of ‘**n**’ lines, holding the gem.

### Constraints

* The number **n will be a positive odd integer between 3 and 59**, inclusive.
* The type of the gem will be a symbol from the standard ASCII table.
* Allowed working time for your program: 0.1 seconds.
* Allowed memory: 16 MB.

### Examples

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |
| 5  \* | --\*--  -\*\*\*-  \*\*\*\*\*  -\*\*\*-  --\*-- | 7  @ | ---@---  --@@@--  -@@@@@-  @@@@@@@  -@@@@@-  --@@@--  ---@--- |  | 13  a | ------a------  -----aaa-----  ----aaaaa----  ---aaaaaaa---  --aaaaaaaaa--  -aaaaaaaaaaa-  aaaaaaaaaaaaa  -aaaaaaaaaaa-  --aaaaaaaaa--  ---aaaaaaa---  ----aaaaa----  -----aaa-----  ------a------ |

## \*\* Decrypt the Messages

You are working for a company which is very concerned about its information and communication. For this reason, they have invented an internal approach to communication between different departments – they are communicating to each other via **messages, which are reversed (written backwards) and then encrypted**. In order to be read and understood, each message has to be decrypted. Your task is to write a program, which **receives all encrypted messages** in a specific communication, **decrypts them** and **prints all decrypted messages at the console as well as the total number of messages** that have been received.

**At the beginning** of a communication, you will receive either the keyword “**START**” (upper case) or “**start**” (lower case), which indicates that you will **start receiving reversed and encrypted messages**. At the end of the communication, you will receive either the keyword “**END**” (upper case) or “**end**” (lower case), which indicates that the communication is over and you need to **show the decrypted messages’ content and total count**. Any **non-empty string** between the “start” and “end” keywords is considered a message. If **no messages have been received** between the “**start**” and the “**end**” keywords, you should print on the console: **“No message received.”**

All messages are case-sensitive and consist of **letters**, **digits,** as well as **some special characters** – ‘**+**’, ‘**%**’, ‘**&**’, ‘**#**’ and ‘**$**’. Letters **from A to M** are **converted** to letters **from N to Z** (A 🡪 N; B 🡪 O; … M 🡪 Z) and letters **from N to Z** are **converted** to letters **from A to M** (N 🡪 A; O 🡪 B; … Z 🡪 M). The **converted** letter should keep the **case** of the **original** letter. The **special characters** are converted in the following way: ‘**+**’ is converted to a **single space** (**‘ ’**), ‘**%**’ is converted to a **comma** (**‘,’**), ‘**&**’ is converted to a **dot** (**‘.’**), ‘**#**’ is converted to a **question mark** (**‘?’**) and ‘**$**’ is converted to an **exclamation mark** (**‘!’**). The **digits** (0-9) are **not converted** and stay the same.

For example, you receive the following message – “**$1+rtnffrz+greprF**” and you start decrypting it. Convert the 1st character ‘**$**’ to ‘**!**’, then the 2nd character – ‘**1**’ stays the same, then covert the 3rd character – ‘**+**’ to single space ‘ ’, ‘**r**’ 🡪 ’**e**’, ‘**t**’ 🡪 ‘**g**’, ‘**n**’ 🡪 ‘**a**’, ‘**f**’ 🡪 ‘**s**’, ‘**f**’ 🡪 ‘**s**’, ‘**r**’ 🡪 ’**e**’ , ‘**z**’ 🡪 ’**m**’, ‘**+**’ 🡪 ‘ ’, ‘**g**’ 🡪 ‘**t**’, ‘**r**’ 🡪 ’**e**’ , ‘**e**’ 🡪 ’**r**’ , ‘**p**’ 🡪 ’**c**’ , ‘**r**’ 🡪 ’**e**’ , ‘**F**’ 🡪 ’**S**’. After decrypting all letters, the message is: “**!1 egassem terceS”** and when you reverse it, you get the original message: “**Secret message 1!**”

### Input

The input data should be read from the console. The input will contain a random number of lines. The line that holds the **keyword “START” or “start”** will always be before the line that holds the **keyword “END” or “end”**. The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

The output data should be printed on the console.

* On the **first line** print the total number of messages that have been received in format: “**Total number of messages: N**” – where N is the number of received and decrypted messages.
* On the next N lines print the decrypted messages.
* If **no messages have been received** between the “**start**” and the “**end**” keywords, you should **print on the console** only one line holding: “**No message received.**”

### Constraints

* The **number of messages** between the “**start**” and the “**end**” keywords will be between 0 and 100.
* The **length of each message** will be between 1 and 1000 symbols.
* Each encrypted message will contain only Latin letters, digits and the special symbols described above.
* Allowed working time: 0.1 seconds. Allowed memory: 16 MB.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Comments** | **Output** |
| START  $$$byyrU  END | We start conversion from the 1st character: $ 🡪 !, $ 🡪 !, $ 🡪 !, b 🡪 o, y 🡪 l, y 🡪 l, r 🡪 e, U 🡪 H and reverse the newly received string “!!!olleH” to the original message “Hello!!!” | Total number of messages: 1  Hello!!! |

|  |  |  |
| --- | --- | --- |
| **Input** | **Comments** | **Output** |
| start  tsrqpon  1rtnFFrz  end | We start conversion from the 1st character: t 🡪 g, s 🡪 f, r 🡪 e, q 🡪 d, p 🡪 c, o 🡪 b, n 🡪 a and reverse the newly received string “gfedcba” to the original message “abcdefg”.  Then we do the same for the second message. | Total number of messages: 2  abcdefg  meSSage1 |

|  |  |  |
| --- | --- | --- |
| **Input** | **Comments** | **Output** |
| start  END | There is no message received. | No message received. |

|  |
| --- |
| **Input** |
| Normal communication message.  START  $rtnffrz+tavjbyybs+rug+gclepar+bg+leg+%rfnryC  end |
| **Output** |
| Total number of messages: 1  Please, try to encrypt the following message! |

## \*\* Bits at Crossroads

Bits are usually very boring. They walk only left <-> right and up <-> down. Your task is to write a program which builds diagonal roads to break the monotonous bits’ habits.

You are given a **square board** of bits (**size NxN**). Bit positions on **each line** are counted from **right to left**. Line numbers are counted from **top to bottom**. Initially all bits are **set to zero**. You can build **two diagonal roads** easily if you know the coordinates of the **roads’ intersection** (line number, bit position). A crossroad is an intersection between two roads.

Example: The line number is 2 and the bit position is 5: (2, 5). There are two diagonal roads – from (0, 7) to (7, 0) and from (0, 3) to (4, 7) and one crossroad (2, 5) (*see Fig. 1*). Cells shaded grey are the roads and cells shaded black are crossroads.

Example 2: We have two predefined crossroads (2, 5) and (3, 2). Now there are 4 crossroads in total – the start points and two additional points (1, 4) and (4, 3) *(see Fig. 2)*.

Your task is to write a program that prints the integer representation of each row from the final board and finds the number of crossroads on the board.

*Fig. 1 Fig. 2*

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Number |
| 0 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 136 |
| 1 | 0 | **1** | 0 | **1** | 0 | 0 | 0 | 0 | 80 |
| 2 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 32 |
| 3 | 0 | **1** | 0 | **1** | 0 | 0 | 0 | 0 | 80 |
| 4 | **1** | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 136 |
| 5 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 4 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 2 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 1 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Number |
| 0 | **1** | 0 | **1** | 0 | **1** | 0 | 0 | 0 | 168 |
| 1 | 0 | **1** | 0 | **1** | 0 | 0 | 0 | **1** | 81 |
| 2 | 0 | 0 | **1** | 0 | **1** | 0 | **1** | 0 | 42 |
| 3 | 0 | **1** | 0 | **1** | 0 | **1** | 0 | 0 | 84 |
| 4 | **1** | 0 | 0 | 0 | **1** | 0 | **1** | 0 | 138 |
| 5 | 0 | 0 | 0 | **1** | 0 | **1** | 0 | **1** | 21 |
| 6 | 0 | 0 | **1** | 0 | 0 | 0 | **1** | 0 | 34 |
| 7 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | **1** | 64 |

### Input

* On the first line, you are given an integer number N that represents the size of the board.
* Each of the next lines will hold the position of a predefined crossroad – two integer numbers, separated with a single space:
  + The first integer will be the line number.
  + The second integer will be the bit position.
* When you read the “**end**” command from the console print the result.

The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

The output data must be printed on the console.

* On the first N lines print the integer representations of each row of the board.
* On the last line print the total count of all crossroads on the board.

### Constrains

* **The size N of the board** is an integer in the range [3 ... 32].
* Each **start point** will **always** be a **zero bit**.
* Each **start point** will **always** be a **valid crossroad - the line number and bit position will both be in the range [0 … N)**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 10  3 1  0 1  5 2  end | 146  77  45  19  47  76  154  305  608  192  4 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 16  2 5  3 2  8 5  12 3  end | 41128  20561  10282  5205  2698  1301  682  1361  2208  4433  8874  17684  35338  5141  10274  20545  14 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 8  2 5  3 2  end | 168  81  42  84  138  21  34  65  4 |