# More Exercises: Strings and Text Processing

Problems for exercises and homework for the [“Programming Fundamentals Extended” course @ SoftUni](https://softuni.bg/courses/programming-fundamentals).

Check your solutions here: <https://judge.softuni.bg/Contests/443>.

## Value of a String

Write a program which finds the **sum** of the **ASCII** **codes** of the **letters** in a given **string**. Your tasks will be a bit harder, because you will have to find the **sum** of **either** the **lowercase** or the **uppercase** letters.

On the **first** line, you will receive the **string**.

On the **second** line, you will receive **one of two possible inputs**:

* If you receive “**UPPERCASE**” 🡺 find the **sum** of **all** **uppercase** **English** **letters** in the previously received string
* If you receive “**LOWERCASE**” 🡺 find the **sum** of all **lowercase** **English** **letters** in the previously received string

You should **not** sum the **ASCII** codes of any characters, which is **not** letters.

At the end print the sum in the following format:

* The total sum is: {sum}

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| HelloFromMyAwesomePROGRAM  LOWERCASE | The total sum is: 1539 |
| AC/DC  UPPERCASE | The total sum is: 267 |

## Diamond Problem

In the programming languages, which permit multiple inheritance, the diamond problem is a very common problem. In our task, the diamond problem is a bit more… money driven.

Your task is to write a program, which **finds** all **diamonds** in a string and **calculates** the **carats** of each diamond.

Each diamond will start with the character ‘<’. After that, it will be followed by **several** **random characters** (contents of the diamond). The diamond will **end** with the character ‘>’.

The **carat value** of the diamond is equal to the **sum of all the digits** in the **contents** of the diamond.

Example: “<2big32diamond>” 🡺 **2 + 3 + 2** 🡺 **7** carats

If the given string contains one or more diamonds, print for each found diamond the following output:

* Found {caratValueOfTheDiamond} carat diamond

If in the given string cannot be found any diamond, print:

* Better luck next time

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| empty**<2big32diamond>**useless**<1another02Diamond>** | Found 7 carat diamond  Found 3 carat diamond |
| No>diamonds<here | Better luck next time |

## Serialize String

You have been tasked to serialize a string. The serialization is done in a special way, in which a character from that string is saved with the indexes at which it is found.

The input will consist of a single input line, containing a single string, which may consist of **ANY ASCII** character. Your task is to serialize the string in the following way:

{char}:{index1}/{index2}/{index3}

The char will be the **character**, and the indexes, will be the **indexes** it is **found** at in the **string**.

**Note:** This problem is a **string problem**, and should **ONLY** use **strings** in its **solution**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| abababa | a:0/2/4/6  b:1/3/5 |
| avjavamsdmcalsdm | a:0/3/5/11  v:1/4  j:2  m:6/9/15  s:7/13  d:8/14  c:10  l:12 |

## Deserialize String

Write a program, which takes the **output** from the **previous task** and turns it back into a **string**.

Until you receive the line “**end**”, you will receive several lines of input on the console, in the following format:

* “{letter}:{index1}/{index2}/{index…}/{indexN}”

Your task is to take every **letter** and its **index** and **form a string** out of them.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| a:0/2/4/6  b:1/3/5  end | abababa |
| a:0/3/5/11  v:1/4  j:2  m:6/9/15  s:7/13  d:8/14  c:10  l:12  end | avjavamsdmcalsdm |

## String Commander

Strings can be hard to manipulate because of their immutable nature. As a master of strings, you have to write a program, which will help the less experienced manipulate them.

On the **first** line, you will receive the **string**, which you have to **manipulate**.

On the **next** input lines, **until** you receive the command "**end**", you’ will receive a **series** of commands in **one** of the **following** formats:

* "Left {count} times" – this command moves **all** elements left count times. On each roll, the **first** element is placed at the **end** of the string.
* "Right {count} times" – this command moves **all** elements left count times. On each roll, the **last** element is placed at the **beginning** of the string.
* "Insert {index} {string}" – **insert** the given string at the index.
* "Delete {startIndex} {endIndex}" – delete the element from the startIndex (**inclusive**) to the endIndex (**inclusive**)

At the end, **print** the **string** after all **modifications**.

### Input

* The first input line will hold **the string**, which we have to manipulate.
* The next lines will hold **commands** in the described formats.
* The input ends with the keyword **"end"**.

### Output

* After receiving the "**end**" command, print the **string** after **all** manipulations.

### Constraints

* All **commands**, **indices** and **counts** will be in the **correct** format and **inside** the **string**. You do **not** have to **check** them **explicitly**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| The Lone Ranger  Delete 0 7  Insert 0 Power  Insert 12 s  end | Power Rangers |
| ReverseItAll  Left 20  Right 83  Delete 0 2  end | ReverseIt |

## Stateless

You will be **given groups** of **2 strings**, each on a **new line**. There will **ALWAYS** be at **least 2 input lines**, and there will **NEVER** be a case when there **are less than 2 input strings**, for a **given element of the input**.

Now to the main logic – the **elements of the input**. You can **refer** to the elements of the input as states.

Each state also has a fiction – the collapsing factor. Your task is to **collapse** **each** **state**, by its **given** **fiction**.

The collapsing is done by **removing all occurrences** of the **fiction** in the **state**, and after that – **removing** the **first** and **last element** of the **fiction**. You must then **repeat the process**, until the **fiction’s length** becomes 0.

When you finish the process, you must **print what is left** from the **state**. If the state is also empty, you should print “(void)”. **NOTE**: **Border spaces** should be **removed**.

Both the **state** and the **fiction** are **strings**, and will be **given each** on a **separate line**. You must read **sequences** of **DOUBLE lines**, and **print** the **result** from the **collapsing**, until you receive the command “collapse”.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| astalavista baby  aaa  aaaa  aa  this will be funny rhight  this  collapse | stlvist bby  (void)  will be funny rght |
| bow chicka mow wow  mow  ahaia  hai  collapse | bw chicka ww  (void) |

## Pyramidic

You will be **given N** – an **integer**. On the next **N input lines**, you will be given **N strings**, which may consist of **any ASCII character**.

Your task is to find the **BIGGEST** **pyramid formation** of **occurrences** of a **SINGLE CHARACTER**, **throughout** **the strings**.

The pyramid is formed by **finding** a **character** on a line, then **finding** **3 consecutive** (**next to each, other**) occurrences of the **same character** on the **next line**, then finding **5 consecutive** occurrences on the next line and so on. . .

Example:

|  |
| --- |
| abacd bbbcd bbbbb |

Result:

|  |
| --- |
| b bbb bbbbb |

Check the examples for more info.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  asdfghjkl  asdgggjkl  asgggggkl  agggggggl  ggggggggg | g  ggg  ggggg  ggggggg  ggggggggg |
| 7  abcdefg  aaadc\\  cbaaaaa  d  ddddasd  !!ddddd!!!!!!!!...  dddddddd | d  ddd  ddddd  ddddddd |

## Nilapdromes \*

Nilapdromes are similar to palindromes, but are quite different. **Nilapdromes** are words which have 1 substring of random characters in the middle, called – the **core**, and **2 identical substrings**, surrounding it, called – the **borders**.

Examples of **nilapdromes** are: “aba”, “asdthisasd”, “baumyaubau”. . .

Examples of **INCORRECT** **nilapdromes** are: “abbc”, “SDSD”, “\_,#$x$#,\_y”.

For example, the **nilapdrome** “baumyaubau” – the **core** is “myau” and the **borders** are “bau”.

You will be receiving input lines, containing **exactly one** nilapdrome, each, until you receive the command “end”.

Your task is to **make**, **from each nilapdrome** – a **new nilapdrome**, with **borders** – **equal** to the **core** (**middle substring**) of the **given one**, and **core** – **equal** to the **borders** of the **given one**.

For example, the **nilapdrome** “baumyaubau” should **result** in “myaubaumyau”.

You should **print** **each result nilapdrome**, after you’ve created it, **BEFORE** reading the **next one.**

**INCORRECT** **nilapdromes**, should be **IGNORED**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| aba  asdthisasd  baumyaubau  end | bab  thisasdthis  myaubaumyau |
| everythingnothingeverything  invalid  donenodedonee  abbc  sdsd  ssdd  dssd  end | nothingeverythingnothing  ssdss |