# Exam Preparation

Part I: [judge.softuni.bg/Contests/906](https://judge.softuni.bg/Contests/906/Programming-Fundamentals-Retake-Exam-05-January-2018-Part-I)

Part II: [judge.softuni.bg/Contests/912](https://judge.softuni.bg/Contests/912/Programming-Fundamentals-Retake-Exam-05-January-2018-Part-II)

# Problem 1 – Snowballs

Tony and Andi love playing in the snow and having snowball fights, but they always argue which makes the best snowballs. Because they are girls (which means they are completely illogical), they have decided to involve you in their fray, by making you write a program which calculates snowball data, and outputs the best snowball value.

You will receive N – an **integer**, the **number** of **snowballs** being made by Tony and Andi.  
**For each snowball** you will receive **3 input lines**:

* On the **first line** you will get the snowballSnow – an **integer**.
* On the **second line** you will get the snowballTime – an **integer**.
* On the **third line** you will get the snowballQuality – an **integer**.

**For each snowball** you must **calculate**its snowballValue by the following formula:

(snowballSnow / snowballTime) ^ snowballQuality

At the end you must print the **highest**calculated snowballValue.

## Input

* On the **first input line** you will receive **N** – the **number** of **snowballs**.
* On the **next N \* 3 input lines** you will be receiving **data** about **snowballs**.

## Output

* As output you must print the **highest** calculated snowballValue, by the formula, **specified above**.
* The output format is:   
  {snowballSnow} : {snowballTime} = {snowballValue} ({snowballQuality})

## Constraints

* The **number** of **snowballs** (N) will be an **integer** in **range [0, 100]**.
* The snowballSnow is an **integer** in **range [0, 1000]**.
* The snowballTime is an **integer** in **range [1, 500]**.
* The snowballQuality is an **integer** in **range [0, 100]**.
* Allowed working **time** / **memory**: **100ms** / **16MB**.

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  10  2  3  5  5  5 | 10 : 2 = 125 (3) |
| 3  10  5  7  16  4  2  20  2  2 | 10 : 5 = 128 (7) |

# Problem 2 – Snowmen

The Snowmen Fray has been on since the dawn of time. Professor Olaf is currently studying the ancient war methods, and needs you to write a program that simulates a battle between snowmen.

You will receive a **sequence** of **integers**, **separated** by **spaces** – the **snowmen**,   
which are **indexed** from **0** – **sequence.length – 1**.

You must **traverse the elements**, from the **first** till the **last**. Each element is an **attacker index**, and its **integer value** is its **target index**. If the **integer value** is **greater than** the **length** of the **sequence**, **divide** it **modulo** (**%**) by the length of the sequence. **Example**: **target** = **12**, **length** = **7**, **target** = **12 % 7** - >**5**.

When you extract **attacker index** and the **target index**, you must calculate the **absolute value** of the **difference** between them.

* If the **difference** is an **even number**, the **attacker wins**.
* if the **difference** is an **odd number**, the **target wins**.
* If the **attacker** and the **target** are **equal** (they are **the same indexes**), that means that snowman **suicides**.

If an element **loses** a battle or **suicides**, it **should NOT be able** to **attack**. It **can** still be a **target** though.

When you finish traversing the whole sequence, you must **remove all elements** that have **lost** or **suicided**, and then you must **start over**. The process must be **repeated** until there is **1snowman left** in the **sequence**.

For each **attacker** and **target**, you must print “{attacker} x {target} -> {winner} wins”.   
The **attacker** and the **target** are **indexes**, and the winner is the **index** of the **winner**.

In that case you must print “{attacker} performed harakiri”.

## Input

* As input you will receive a **single input line** containing the **sequence** of **integers**, **separated** by **spaces**.

## Output

* As output you must print each action between attacker and target, in the formats specified above.

## Constraints

* The **elements** of the **sequence** will be **between 0** and **100**.
* The **integers** in the **sequence** will be in **range [0, 1000]**.
* Allowed working **time** / **memory**: **100ms** / **16MB**.

## Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 4 3 2 1 0 | 0 x 4 -> 0 wins  1 x 3 -> 1 wins  2 performed harakiri  0 performed harakiri | First: Attacker – 0, Target – 4. Difference = 4 (even) Attacker wins, Target lost. Second: Attacker – 1, Target – 3. Difference = 2 (even) Attacker wins, Target lost. Third: Attacker – 2, Target – 2. Equal, Attacker suicided. All other elements have lost. We remove them from the sequence. Sequence – 43. Length = 2. First: Attacker – 0, Target – 4 (% 2) = 0. Equal, Attacker suicided.  There is 1 element left, we stop the program. |
| 25 31 6 9 2 4 7 | 0 x 4 -> 0 wins  1 x 3 -> 1 wins  2 x 6 -> 2 wins  5 x 4 -> 4 wins  0 x 1 -> 1 wins  1 performed harakiri |  |

# Problem 3 – Snowflake

Tony and Andi have been researching the snowflake structure for quite some time now. They managed to write a program which extracts a text pattern from the snowflake, but it oftenly produces invalid data. They begged you to help them, so you must write a program which validates the text pattern.

The **snowflake** has **3 elements**:

* **Surface** – should **NOT** contain **letters** and **digits**.
* **Mantle** – should **ONLY** contain **digits** and **underscores** (“\_”).
* **Core** – should **ONLY** contain **letters**.

You will receive several input lines, which will represent the **snowflake**. A **valid** snowflake is in the following format:

{surface}  
{mantle}  
{surface}{mantle}{core}{mantle}{surface}  
{mantle}  
{surface}

You must check if **all elements** are **valid**, by the **rules specified above**. If even one element is invalid, you should print “**Invalid**”.

If all are valid you should print “**Valid**” and the **length** of the core part on the **next line**.

## Input

* The input will consist of exactly **5 input lines**, in the format **specified above**.

## Output

* If **all the elements** of the **snowflake** are **valid**, you must print “**Valid**”, and the **length** of the **core**.
* If **even one** of the **elements** is **invalid**, you must print “**Invalid**”.

## Constraints

* The elements of the input may contain **ANY ASCII character**.
* The **input** will **always consist** of **5 input lines**.
* The input **may not always** be in the **valid snowflakeformat**.
* Allowed working **time** / **memory**: **100ms** / **16MB**.

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| \*-\*  1\_1\_1  \*\*\*444asd444\*\*\*  1\_1\_1  \*-\* | Valid  3 |
| !!!!!!  14741  -2Asdasdasdasd555!  \_\_\_  -- | Valid  12 |
| Asd  Asd  Asd  Asd  asd | Invalid |

# Problem 4 – Snowwhite

Snow White loves her dwarfs, but there are so many and she doesn’t know how to order them. Does she order them by name? Or by color of their hat? Or by physics? She can’t decide, so its up to you to write a program that does it for her.

You will be receiving **several input lines** which contain **data** about **dwarfs** in the following format:

{dwarfName} <:> {dwarfHatColor} <:> {dwarfPhysics}

The dwarfName and the dwarfHatColor are **strings**. The dwarfPhysics is an **integer**.

You must **store** the **dwarfs** in your program. There are several rules though:

* If **2 dwarfs** have the **same name** but **different color**, they should be **considered different dwarfs**, and you should store **both** of them.
* If **2 dwarfs** have the **same name** and the **same color**, **store** the **one** with the **higher physics**.

When you receive the command “Once upon a time”, the input ends. You must **order** the **dwarfs** by **physics** in **descending order** and thenby **totalcount** of **dwarfs** with the **same hat color** in **descending order**.   
Then you must print them all.

## Input

* The input will consists of **several input lines**, containing **dwarf data** in the format, specified above.
* The input **ends** when you receive the command “Once upon a time”.

## Output

* As output you must print the **dwarfs**, **ordered** in the way , specified above.
* The output format is: ({hatColor}) {name} <-> {physics}

## Constraints

* The dwarfName will be a **string** which may contain **any ASCII** character except ‘’ (space), ‘<’, ‘:’, ‘>’.
* The dwarfHatColor will be a **string** which may contain **any ASCII** character except ‘’ (space), ‘<’, ‘:’, ‘>’.
* The dwarfPhysics will be an **integer** in **range [0, 231 – 1]**.
* There will be **no invalid** input lines.
* If **all sorting criteria fail**, the order should be by **order** of **input**.
* Allowed working **time** / **memory**: **100ms** / **16MB**.

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho <:> Red <:> 2000  Tosho <:> Blue <:> 1000  Gosho <:> Green <:> 1000  Sasho <:> Yellow <:> 4500  Prakasho <:> Stamat <:> 1000  Once upon a time | (Yellow) Sasho <-> 4500  (Red) Pesho <-> 2000  (Blue) Tosho <-> 1000  (Green) Gosho <-> 1000  (Stamat) Prakasho <-> 1000 |
| Pesho <:> Red <:> 5000  Pesho <:> Blue <:> 10000  Pesho <:> Red <:> 10000  Gosho <:> Blue <:> 10000  Once upon a time | (Blue) Pesho <-> 10000  (Blue) Gosho <-> 10000  (Red) Pesho <-> 10000 |