# Problem 1 – Sweet Dessert

Ivancho and his girlfriend are **throwing a party**. She plans **to cook her favorite dessert**. She asks Ivancho to **buy** the **needed products**. The **number of desserts** depends on **how many people will be coming**. She can prepare the dessert **in portions of 6**. If there are **5 guests** coming, she will still **cook 6 portions,** **10 guests – cook 12**. The products for the dessert are **bananas, eggs and berries**. For **a set of 6** she needs **2 bananas, 4 eggs and 0.2 kilos berries**.

You will be given **the amount of money Ivancho has**, the **number of guests** and the **prices of the products**. You have to help Ivancho **calculate** if the **cash** he has is **enough to buy all of the products**, or how much more money he needs.

### Input

The input data should be read from the console. It will consist of **exactly 5 lines**:

* The **amount of cash** Ivancho has – **floating-point number** in **range [0.00…1,000,000,000.00]**
* The **number of guests – integer in range [0…1,000,000,000]**
* The **price of bananas** for a **single unit – floating-point number** in **range [0.00…1,000.00]**
* The **price of eggs** for a **single unit – floating-point number** in **range [0.00…1,000.00]**
* The **price of berries** for a **kilo – floating-point number** in **range [0.00…1,000.00]**

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.

* **If the calculated price of the products is less or equal to the money Ivancho has:**
  + “Ivancho has enough money - it would cost {the cost of the products}lv.”
* **If the calculated price of the products is more than the money Ivancho has:**
  + “Ivancho will have to withdraw money - he will need {neededMoney}lv more.”
* **All prices** must be **rounded to two digits after the decimal point.**

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 10  12  0.35  0.20  4.50 | Ivancho has enough money - it would cost 4.80lv. | For 12 guests – 2 sets of 6 portions  Needed product:  2\*(2 bananas), 2\*(4 eggs), 2\*(0.2 kilos berries)  2\*(2\*0.35) + 2\*(4\*0.20) + 2\*(0.2\*4.50) = 4.80  4.80 <= 10 – the money will be enough. |
| **Input** | **Output** | **Comments** |
| 20  33  0.60  0.50  10 | Ivancho will have to withdraw money - he will need 11.20lv more. | For 33 guests – 6 sets of 6 portions  Needed product:  6\*(2 bananas), 6\*(4 eggs), 6\*(0.2 kilos berries)  6\*(2\*0.60) + 6\*(4\*0.50) + 6\*(0.2\*10.00) = 31.20  31.20 > 20 – need 11.20 lv. more. |

# Problem 2 – Array Manipulator

Trifon has finally become a junior developer and has received his first task. It’s about manipulating an array of integers. He is not quite happy about it, since he hates manipulating arrays. They are going to pay him a lot of money, though, and he is willing to give somebody half of it if to help him do his job. You, on the other hand, love arrays (and money) so you decide to try your luck.

The array may be manipulated by one of the following commands

* **exchange {index}** – splits the array **after** the given index, and exchanges the places of the two resulting sub-arrays. E.g. [1, 2, 3, 4, 5] -> exchange 2 -> result: [4, 5, 1, 2, 3]
  + If the index is outside the boundaries of the array, print “**Invalid index**”
* **max** **even/odd**– returns the **INDEX** of the max even/odd element -> [1, 4, 8, 2, 3] -> **max odd** -> print **4**
* **min** **even/odd** – returns the **INDEX** of the min even/odd element -> [1, 4, 8, 2, 3] -> **min even** > print **3**
  + If there are two or more equal **min/max** elements, return the index of the **rightmost** one
  + If a **min/max even/odd** element **cannot** be found, print **“No matches”**
* **first {count}** **even/odd**– returns the first {count} elements -> [1, 8, 2, 3] -> **first 2 even** -> print [**8, 2]**
* **last {count}** **even/odd** – returns the last {count} elements -> [1, 8, 2, 3] -> **last 2 odd** -> print [**1, 3]**
  + If the count is greater than the array length, print “**Invalid count**”
  + If there are **not** **enough** elements to satisfy the count, print as many as you can. If there are **zero** **even/odd** elements, print an empty array “[]”
* **end** – stop taking input and print the final state of the array

### Input

* The input data should be read from the console.
* On the first line, the initial array is received as a line of integers, separated by a single space
* On the next lines, until the command “**end**” is received, you will receive the array manipulation commands
* 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.
* On a separate line, print the output of the corresponding command
* On the last line, print the final array in **square brackets** with its elements separated by a comma and a space
* See the examples below to get a better understanding of your task

### Constraints

* The **number of input lines** will be in the range [2 … 50].
* The **array elements** will be integers in the range [0 … 1000].
* The **number of elements** will be in the range [1 .. 50]
* The **split index** will be an integer in the range [-231 … 231 – 1]
* **first/last count** will be an integer in the range [1… 231 – 1]
* There will **not** be redundant whitespace anywhere in the input
* Allowed working time for your program: 0.1 seconds. Allowed memory: 16 MB.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 3 5 7 9  exchange 1  max odd  min even  first 2 odd  last 2 even  exchange 3  end | 2  No matches  [5, 7]  []  [3, 5, 7, 9, 1] |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 10 100 1000  max even  first 5 even  exchange 10  min odd  exchange 0  max even  min even  end | 3  Invalid count  Invalid index  0  2  0  [10, 100, 1000, 1] |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 10 100 1000  exchange 3  first 2 odd  last 4 odd  end | [1]  [1]  [1, 10, 100, 1000] |

# Problem 3. Football Standings

**You will be given** information about **results of football matches**. **Create a standings table by points.** For **win the team gets 3** **points**, **for loss – 0** and **for draw – 1**. Also **find the top 3 teams with most scored goals in descending order**. If **two or more teams** are with **same goals scored** or **same points** **order them by name** in **ascending order**.

The name of each team is **encrypted**. You must **decrypt it before** **proceeding with calculating** statistics. You will be given some **string key** and the **team name will be placed between that key in reversed order.**

**For example:** the **key**: “???”;

**String to decrypt**: “kfle???airagluB???gertIt%%” -> “airagluB” -> “Bulgaria”

Also you should **ignore the letter casing** in the team names. **For example:**

buLgariA = BulGAria = bulGARIA = BULGARIA

## Input / Constrains

* On the **first line** of input you will get the **key** that will be used for decryption
* On the next lines until you receive “**final**” you will get lines in format:

**{encrypted teamA} {encrypted teamB} {teamA score}:{teamB score}**

* **Team scores** will be **integer numbers** in the **range [0...231]**

## Output

**League standings:**

**{place}. {TEAM NAME} {points}**

**...**

* **Print the standings table** **ordered descending by points** in format:

Where **place** is a number in range **[1… number of teams].**

**Top 3 scored goals:**

**- {team name} -> {goals}**

**- {team name} -> {goals}**

**- {team name} -> {goals}**

* Then you should print the top 3 team **ordered by goals in descending order** in format:
* **All team’s names** should be **uppercase.**
* **For more clarification, see the examples on the next page.**

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ??  ??ecnarF?? ??kramneD?? 0:0  ..??airagluB??32 ??dnalgnE??gf 3:2  Fg??NIAPS?? fgdrt%#$??YNAMREG??gtr 3:4  ??eCnArF?? >>??yLATi??<< 2:2  final | League standings:  1. BULGARIA 3  2. GERMANY 3  3. FRANCE 2  4. DENMARK 1  5. ITALY 1  6. ENGLAND 0  7. SPAIN 0  Top 3 scored goals:  - GERMANY -> 4  - BULGARIA -> 3  - SPAIN -> 3 |
| **Input** | **Output** |
| KZL  fdKZLairagluBKZL KZLkramneDKZLll 2:0  kzljjjKZLAiRaGluBKZL KZLylATIKZLkk 1:1  KZLkRamnedKZL KZLYlatiKZL 4:4  final | League standings:  1. BULGARIA 4  2. ITALY 2  3. DENMARK 1  Top 3 scored goals:  - ITALY -> 5  - DENMARK -> 4  - BULGARIA -> 3 |

# Problem 4 – Cubic’s Messages

Cubic is a veteran soldier from The Great Cubic Army. He has even participated in the Spherical Invasion as a Sergeant First Class. As a veteran, Cubic has some personal security issues – he communicates only trough text messages and sends them in a specific encrypted way, which you must decrypt in order to understand what he is saying.

You will begin receiving lines of input, which will consist of random ASCII characters – Cubic’s encrypted lines. After each line you will receive a number – the length of the message he sent. Cubic might send false messages, in an act to confuse his “enemies”. You must capture only the messages that follow a certain format.

According to that format the **valid** messages:

* Consist of **m** characters, where **m** is the integer entered after each encrypted line.
* Has only digits before itself in the encrypted line
* Consists only of English alphabet letters
* Has no English alphabet letters after itself in the encrypted line

**Any** message that **does not follow** the, specified above, rules, is **invalid**, and you must **ignore it**.

After you find **all valid** messages, you need to find their **verification code**. Every message has its own verification code, which Cubic gives in order to verify the message. **Take all the digits before the message** and all the digits **after the message** and consider them as **indexes**. If they are **valid existing** indexes **in the message**, **form a string** with those indexes **taking characters from the message**. If an index is **nonexistent**, put a **space** there. The string you form up is the verification code for the current message.

### Input

* The input will always come in the form of 2 lines, except when it is the line terminating the input sequence.
* The first input line will contain random ASCII characters, and the second – a number.
* When the line “**Over!**” is entered, the input sequence ends.

### Output

* The output is simple. You must print all the valid messages you’ve found, each on a new line, and their verification codes, if they have such.
* The format of output is “**{message} == {verificationCode}**”.

### Constraints

* The input lines can consist of **ANY ASCII** character.
* There will be **NO** such cases as an encrypted message without a number before it.
* The number will be a valid integer in the range [0, 100].
* Allowed time/memory: 100ms/16MB

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| **1234test4321**  **4**  **0000oooo0000**  **4**  **Over!** | **test == est tse**  **oooo == oooooooo** |

|  |  |
| --- | --- |
| **Input** | **Output** |
| **1wat!**  **3**  **#23asd33**  **3**  **333asd3a**  **3**  **100dun2**  **3**  **Over!** | **wat == a**  **dun == uddn** |