More Exercises: Dictionaries and Lists

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

Check your solutions here: https://judge.softuni.bg/Contests/582.

Problem 1. Sort Times

Write a program, which receives a **list of times** (space-separated, 24-hour format) and **sorts** them in **ascending order**. Print the sorted times **comma-separated**.

Example: **06:55, 02:30, 23:11 ➔ 02:30, 06:55, 21:11**

Examples

Input	Output
`00:00 06:04 02:59 10:33 11:22 06:01`	`00:00, 02:59, 06:01, 06:04, 10:33, 11:22`
`04:25 04:21 04:19`	`04:19, 04:21, 04:25`
`00:00 23:59 12:00 16:00`	`00:00, 12:00, 16:00, 23:59`

Problem 2. Odd Filter

Write a program, which receives an array of **integers** (space-separated), **removes** all the odd numbers, then **converts** the remaining numbers to **odd numbers**, based on these conditions:

- If the number is **larger than** the **average** of the collection of remaining numbers, **add 1** to it.
- If the number is **smaller than** the **average** of the collection of remaining numbers, **subtract 1** from it.

After you convert all of the elements to odd numbers, **print** them on the console **(space-separated)**.

Examples

Input	Output
`1 2 3 4 5 6 7 8 9 10`	`1 3 5 9 11`
`99 88 77 66 55 4 33 22 11`	`89 67 3 21`
`23 32 199 723 8127 95`	`31`

Problem 3. Immune System

An **organism** can encounter different types of **viruses**. It stores them in its **immune system**. If it has already encountered the virus, it fights it **faster** than if it hasn’t encountered it yet.

The immune system can calculate the **virus’ strength** before it fights it. It is the **sum** of **all the virus name’s letters’ ASCII codes, divided by 3**.

The immune system can also **calculate** the time it takes to **defeat** a **virus** in **seconds**. It is equal to the **virus strength, multiplied** by the **length** of the virus’ **name**.

When you calculate the **time to defeat** the virus, **convert** it to **minutes** and **seconds** (500 ➔ 8m 20s). **Do not** use any leading zeroes for the minutes and seconds.

The virus is **fought** according to **these conditions**:

- If the immune system **defeats** the virus, print: "`{virusName} defeated in {virusDefeatMinutes}m {virusDefeatSeconds}s.`"
- If the virus' strength is **more than** the **immune system's strength**, print "`Immune System Defeated.`" and exit the program.

After a virus is **defeated**, the **immune system** regains **20%** of its **strength**. If the 20 percent **exceeds** the **initial health** of the immune system, set it to the **initial health** instead.

Example: The virus "**`flu1`**":

- Virus Strength: **102 (f) + 108 (l) + 117 (u) + 49 (1)** = **376 / 3** = **125.33 = 125**.
- Time to defeat: 125 * 4 (virus name **length**) = 500 seconds ➔ 8m 20s.

Example 2: Encountering "**`flu1`**" a **second time**:

- Time to defeat: **(125 * 4) / 3** = **166.66** ➔ **166 seconds**

If you encounter a virus any subsequent times, **do not** decrease its **time to defeat** further. When you receive the line "**end**", print the status of the immune system in the format "**`Final Health: {finalHealth}`**".

Input

- First line: the **initial health** of the immune system
- On new lines, until you receive "**end**": **virus names**

Output

A **defeated** virus' output looks like this:

- First line: "**`Virus {virusName}: {virusStrength} => {virusDefeatSeconds}`**"
- Second line: "**`{virusName} defeated in {defeatMins}m {defeatSecs}s.`**"
- Third line: "**`Remaining health: {remainingHealth}`**". The remaining health is printed **before** its regeneration.

Examples

Input	Output
5000 flu1 test flu1 virussssssss end	Virus flu1: 125 => 500 seconds flu1 defeated in 8m 20s. Remaining health: 4500 Virus test: 149 => 596 seconds test defeated in 9m 56s. Remaining health: 4404 Virus flu1: 125 => 166 seconds flu1 defeated in 2m 46s. Remaining health: 4834 Virus virussssssss: 419 => 4609 seconds virussssssss defeated in 76m 49s. Remaining health: 391 Final Health: 469
1750 Ebola ebola Ebola end	Virus Ebola: 161 => 805 seconds Ebola defeated in 13m 25s. Remaining health: 945 Virus ebola: 171 => 855 seconds ebola defeated in 14m 15s. Remaining health: 279

	Virus Ebola: 161 => 268 seconds Ebola defeated in 4m 28s. Remaining health: 66 Final Health: 79
5700 wannacry iskaplache wannacry	Virus wannacry: 289 => 2312 seconds wannacry defeated in 38m 32s. Remaining health: 3388 Virus iskaplache: 348 => 3480 seconds iskaplache defeated in 58m 0s. Remaining health: 585 Virus wannacry: 289 => 770 seconds Immune System Defeated.

Problem 4. Supermarket Database

Write a program, which keeps information about **products** and their **prices**. Each product has a **name**, a **price** and its **quantity**. If the product **doesn't exist** in the database yet, **add** it with its **starting quantity**.

If you receive a product, which **already exists** in the database, **increase** its quantity by the input quantity and if its **price** is different, **replace** the price as well.

You will receive products' **names**, **prices** and **quantities** on **new lines**. Until you receive the command "**stocked**", keep adding items to the database. When you do receive the command "**stocked**", print the items with their **names**, **prices**, **quantities** and **total price** of all the products with that name. When you're done printing the items, print the **grand total price** of all the items.

*Note: The **grand total** is calculated, based on the **latest price** of the products.*

Input

- Until you receive "**stocked**", the products come in the format: "**{name} {price} {quantity}**".
- The product data is **always** delimited by a **single space**.

Output

- Print information about **each product**, following the format:
  "**{name}: ${price:F2} * {quantity} = ${total:F2}**"
- On the next line, print **30 dashes**.
- On the final line, print the **grand total** in the following format:
  "**Grand Total: ${grandTotal:F2}**"

Examples

Input	Output
Beer 2.20 100 IceTea 1.50 50 NukaCola 3.30 80 Water 1.00 500 stocked	Beer: $2.20 * 100 = $220.00 IceTea: $1.50 * 50 = $75.00 NukaCola: $3.30 * 80 = $264.00 Water: $1.00 * 500 = $500.00 ------------------------------ Grand Total: $1059.00
Beer 2.40 350 Water 1.25 200 IceTea 5.20 100 Beer 1.20 200	Beer: $1.20 * 550 = $660.00 Water: $1.25 * 200 = $250.00 IceTea: $0.50 * 220 = $110.00 ------------------------------

IceTea 0.50 120 stocked	Grand Total: $1020.00
CesarSalad 10.20 25 SuperEnergy 0.80 400 EvenSupererEnergy 1.00 400 Beer 1.35 350 beer 0.50 450 IceCream 1.50 25 stocked	CesarSalad: $10.20 * 25 = $255.00 SuperEnergy: $0.80 * 400 = $320.00 EvenSupererEnergy: $1.00 * 400 = $400.00 Beer: $1.35 * 350 = $472.50 beer: $0.50 * 450 = $225.00 IceCream: $1.50 * 25 = $37.50 ------------------------------ Grand Total: $1710.00

Problem 5. Parking Validation

SoftUni just got a huge, shiny new **parking lot** in a super-secret location (under the Code Ground hall). It’s so fancy, it even has online **parking validation**. Except, the online service doesn’t work. It can only receive users’ data, but doesn’t know what to do with it. Good thing you’re on the dev team and know how to fix it, right?

Write a program, which validates parking for an online service. Users can **register** to park and **unregister** to leave.

The system supports **license plate validation**. A valid license plate has the following **3** distinct characteristics:

- It is **always exactly 8 characters long**.
- Its **first 2** and **last 2 characters** are always **uppercase Latin letters**
- The **4 characters in the middle** are always **digits**

If any of the aforementioned conditions fails, the **license plate** is **invalid**.

The program **receives 2 commands**:

- “**`register {username} {licensePlateNumber}`**”:
  - The system only supports **one car per user** at the moment, so if a user tries to register **another license plate**, using the **same username**, the system should print:
    “**`ERROR: already registered with plate number {licensePlateNumber}`**”
  - If the **license plate** is **invalid**, the system should print:
    “**`ERROR: invalid license plate {licensePlateNumber}`**“
  - If the user tries to register **someone else’s license plate**, the system should print:
    “**`ERROR: license plate {licensePlateNumber} is busy`**”
  - If the aforementioned checks **pass successfully**, the plate **can be registered**, so the system should print:
    “**`{username} registered {licensePlateNumber} successfully`**”
- “**`unregister {username}`**”:
  - If the user is **not present** in the database, the system should print:
    “**`ERROR: user {username} not found`**”
  - If the aforementioned check passes successfully, the system should print:
    “**`user {username} unregistered successfully`**”

After you execute all of the commands, **print** all the currently **registered users** and their **license plates** in the format:

- “**`{username} => {licensePlateNumber}`**”

Input

- First line: **n** – **number of commands** – **integer**
- Next **n** lines: **commands** in one of **two** possible formats:

- Register: “`register {username} {licensePlateNumber}`”
- Unregister: “`unregister {username}`”

The input will **always** be **valid** and you **do not need** to check it explicitly.

Examples

Input	Output

`5`
`register some0ne CS1234JS`
`register vankata JAVA123S`
`register vankata AB4142CD`
`register housey VR1223EE`
`unregister housey`
	`some0ne registered CS1234JS successfully`
	`ERROR: invalid license plate JAVA123S`
	`vankata registered AB4142CD successfully`
	`housey registered VR1223EE successfully`
	`user housey unregistered successfully`
	`some0ne => CS1234JS`
	`vankata => AB4142CD`

`4`
`register testUser AA4132BB`
`register testuser AA4132BB`
`register testuser AA9999BB`
`unregister testUser`
	`testUser registered AA4132BB successfully`
	`ERROR: license plate AA4132BB is busy`
	`testuser registered AA9999BB successfully`
	`user testUser unregistered successfully`
	`testuser => AA9999BB`

`7`
`register gosho mm1111XX`
`register gosho MM1111xx`
`register gosho MMaaaaXX`
`unregister gosho`
`register gosho MM1111XX`
`unregister gosho`
`unregister pesho`
	`ERROR: invalid license plate mm1111XX`
	`ERROR: invalid license plate MM1111xx`
	`ERROR: invalid license plate MMaaaaXX`
	`ERROR: user gosho not found`
	`gosho registered MM1111XX successfully`
	`user gosho unregistered successfully`
	`ERROR: user pesho not found`

Problem 6. Byte Flip

Write a program, which receives a **string array** (space-separated), containing **bytes** in **hexadecimal format** with the **digits reversed**.

Your task is to **remove** any elements whose length is **different than 2**, then **reverse** the digits in **every number**, and finally **reverse** the whole collection and **convert every element** from **hexadecimal numbers** to **characters** from the **ASCII table**.

Print the resulting string of **ASCII characters** on the console.

Input

- First line: the **array** of **strings**, representing a **byte array**.

Output

- First line: The **resulting string** from the input.

Examples

Input	Output
`A 12 B 46 C 56 DDD 46 EEE F6 FFF 36 56 46`	`decoded!`
`37 56 47 97 26 02 D6 56 86 47 02 07 96 C6 66`	`flip them bytes`
`E7 E7 E7 155 33 F5 C 23 12 13`	`1!2_3~~~`

Problem 7. * Take/Skip Rope

Write a program, which reads a **string** and **skips** through it, extracting a **hidden message**. The algorithm you have to implement is as follows:

Let's take the string "**`skipTest_String044170`**" as an example.

Take every **digit** from the string and **store it** somewhere. After that, **remove** all the digits from the string. After this operation, you should have **two lists of items**: the **numbers list** and the **non-numbers list**:

- Numbers list: **`[0, 4, 4, 1, 7, 0]`**
- Non-numbers: **`[s, k, i, p, T, e, s, t, _, S, t, r, i, n, g]`**

After that, take every digit in the **numbers list** and split it up into a **take list** and a **skip list**, depending on whether the digit is in an **even** or an **odd** index:

- Numbers list: **`[0, 4, 4, 1, 7, 0]`**
- Take list: **`[0, 4, 7]`**
- Skip list: **`[4, 1, 0]`**

Afterwards, **iterate** over both of the lists and **skip** **`{skipCount}`** characters from the **non-numbers list**, then **take** **`{takeCount}`** characters and store it in a **result string**. Note that the skipped characters are **summed up** as they go. The process would look like this on the aforementioned **non-numbers list**:

1. Skip **4** characters (total **0**), take **0** characters ➔ "**skipTest_String**" ➔ Taken: "" ➔ Result: ""
2. Skip **1** characters (total **4**), take **4** characters ➔ "**skipTest_String**" ➔ Taken: "**Test**" ➔ Result: "**Test**"
3. Skip **0** characters (total **9**), take **7** characters ➔ "**skipTest_String**" ➔ Taken: "**String**" ➔ Result: "**TestString**"

After that, just print the **result string** on the console.

Input

- First line: The **encrypted** message as a **string**

Output

- First line: The **decrypted** message as a **string**

Constraints

- The count of digits in the input string will **always be even**.
- The encrypted message will contain any printable ASCII character.

Examples

Input	Output
`T2exs15ti23ng1_3cT1h3e0_Roppe`	`TestingTheRope`
`O{1ne1T2021wf312o13Th111xreve!!@!`	`OneTwoThree!!!`
`this forbidden mess of an age rating 0127504740`	`hidden message`