# Data Transfer

You will be given several lines of **messages containing data**. You have to **check for the validity of the lines**. A **valid** line should be in the format: **“s:{sender};r:{receiver};m--"{message} "”**

* **sender –** could contain **any ascii character except for “;”**
* **receiver –** could contain **any ascii character except for “;”**
* **message –** should contain **only letters and spaces**

In each valid message there is a **hidden size of data transfer.** The size of the data transfer is **calculated by the sum of all digits in the names of the sender and receiver**. After each valid message print a line in the format: “**{senderName} says "{currentMessage}" to {recieverName}”.** The **printed names should contain only letters and spaces**. Example: sender “P@e$5sh#o Go^4sh5ov” is **valid** and **matches**, but when printing his name, **we only print** “Pesho Goshov”.

At the end print a line in the format **“Total data transferred: {totalData}MB”**.

## Input / Constraints

* First line will be a number **n** in range [1, 100].
* The next **n** lines will be **strings**.

## Output

* Print each valid message in the format described above.
* Print the total amount of data transfer.

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  s:P5%es4#h@o;r:G3#o!!s2h#2o;m--"Attack"  s:G3er%6g43i;r:Kak€$in2% re3p5ab3lic%an;m--"I can sing"  s:BABAr:Ali;m-No cave for you | Pesho says "Attack" to Gosho  Gergi says "I can sing" to Kakin repablican  Total data transferred: 45MB |
| 5  s:B^%4i35454l#$l;r:Mo5l#$34l%y;m--"Run"  s:Ray;r:To^^5m;m--"Hidden Message"  bla;r:1234a;m--Hello  s:M#$%$#^6767687654545e;r:Yo54$#@#u5;m--"$$$"  s:M#$@545e;r:You241$@#23;m"Hello" | Bill says "Run" to Molly  Ray says "Hidden Message" to Tom  Total data transferred: 42MB |

# Tagram

You will receive **several input lines** in one of the following formats:

* "{username} -> {tag} -> {likes}"
* "ban {username}"

The username and tag are strings. **Likes** will be an integer number. You need to keep track of **every user**.

When you receive a **user**, a **tag** and **likes**, register the user if **he isn't present**, **otherwise add** the tag and the likes. If the user has already used the tag just add the likes to it.

If you receive **"ban {username}"** and **the username exists**, remove him from the database.

You should end your program when you receive the command "end". At that point you should print the users, **ordered by total likes in desecending order, then ordered by the tags’ count in ascending order**. **Foreach** player print their tag and likes.

## Input / Constraints

* The input comes in the form of commands in one of the formats specified above.
* Username and tag **will always be one word string, containing no whitespaces**.
* Likes will be an **integer** in the **range [0, 1000]**.
* There will be **no invalid** input lines.
* The programm ends when you receive the command "end".

## Output

* The output format for each player is:

"{username}"

"- {tag}: {likes}"

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Katty -> healthy -> 50  Elvin -> food -> 20  John -> music -> 30  Katty -> fitness -> 100  end | Katty  - healthy: 50  - fitness: 100  John  - music: 30  Elvin  - food: 20 |
| **Input** | **Output** |
| Monica -> music -> 100  Monica -> dance -> 50  John -> chill -> 200  Santa -> angry -> 300  ban Santa  Joshua -> football -> 500  end | Joshua  - football: 500  John  - chill: 200  Monica  - music: 100  - dance: 50 |
| Ani -> A1 -> 100  Bobi -> B2 -> 100  Bobi -> BB2 -> 150  Ani -> AA1 -> 100  Ani -> AAA1 -> 50  end | Bobi  - B2: 100  - BB2: 150  Ani  - A1: 100  - AA1: 100  - AAA1: 50 |

# Miner

We get as input **the size** of the **field** in which our miner moves. The field is **always a square**. After that we will receive the commands which represent the directions in which the miner should move. The miner **starts** from position – ‘**s’**. The commands will be: **left**, **right**, **up** and **down**. If the miner has reached a side edge of the field and the next command indicates that he has to get out of the field, he must **remain on his current possition and ignore the current command**. The possible characters that may appear on the screen are:

* **\*** – a regular position on the field.
* **e** – the end of the route.
* **c -** coal
* **s** - the place where the **miner starts**

Each time when the miner finds a coal, he collects it and **replaces it with '\*'**. Keep track of the **count of the collected coals**. If the miner collects all of the coals in the field, the program stops and you have to print the following message: **"You collected all coals! ({rowIndex}, {colIndex})"**.

If the miner **steps at 'e' the game is over (the program stops)** and you have to print the following message: **"Game over! ({rowIndex}, {colIndex})"**.

If there are no more commands and none of the above cases had happened, you have to print the following message: **"{remainingCoals} coals left. ({rowIndex}, {colIndex})"**.

## Input

* **Field size** – an integer number.
* **Commands to move** the miner – an array of strings separated by **" "**.
* **The field: some of the following characters (\*, e, c, s),** separated by whitespace (" ");

## Output

* There are three types of output:
  + If all the coals have been collected, print the following output: **"You collected all coals! ({rowIndex}, {colIndex})"**
  + If you have reached the end, you have to stop moving and print the following line: **"Game over! ({rowIndex}, {colIndex})"**
  + If there are no more commands and none of the above cases had happened, you have to print the following message: "{totalCoals} coals left. ({rowIndex}, {colIndex})"

## Constraints

* The **field size** will be a 32-bit integer in the range [0 … 2 147 483 647].
* The field will always have only one 's'.
* Allowed working time for your program: 0.1 seconds.
* Allowed memory: 16 MB.

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  up right right up right  \* \* \* c \*  \* \* \* e \*  \* \* c \* \*  s \* \* c \*  \* \* c \* \* | Game over! (1, 3) |
| 4  up right right right down  \* \* \* e  \* \* c \*  \* s \* c  \* \* \* \* | You collected all coals! (2, 3) |
| 6  left left down right up left left down down down  \* \* \* \* \* \*  e \* \* \* c \*  \* \* c s \* \*  \* \* \* \* \* \*  c \* \* \* c \*  \* \* c \* \* \* | 3 coals left. (5, 0) |

# Cups and Bottles

You will be given a **sequence of integers** – each indicating a **cup's capacity**. After that you will be given **another sequence of integers** – a **bottle** **with** **water** in it. Your job is to try to **fill up** all of the cups.

Filling is done by picking **exactly one** bottle at a time. You must start picking from **the last received bottle** and start filling from **the first entered cup**. If the current bottle has **N** water, you **give** the **first entered cup N** water and **reduce** its integer value by **N**.

When a cup's **integer value** reaches **0 or less**, it **gets removed**. It is **possible** that the current cup's value is **greater** than the current bottle's value. **In that case** you **pick bottles until** you reduce the cup's integer value to **0 or less**. If a bottle's value is **greater** **or equal to** the cup's **current** value, you fill up the cup and **the remaining water** **becomes wasted**. You should **keep track of the wasted litters of water** and **print it at the end of the program**.

If you **have managed** to **fill up all of the cups**, print the **remaining water bottles**, from the **last entered** **– to the first**, otherwise you must print the **remaining cups**, by **order of entrance** – from the **first entered – to the last**.

### Input

* On the **first line** of input you will receive the integers, representing the **cups' capacity**, **separated** by a **single space**.
* On the **second line** of input you will receive the integers, representing the **filled** **bottles**, **separated** by a **single space**.

### Output

* On the first line of output you must print the remaining bottles, or the remaining cups, depending on the case you are in. Just **keep** the **orders of printing exactly as specified**.
  + **"Bottles: {remainingBottles}"** or **"Cups: {remainingCups}"**
* On the second line print the wasted litters of water in the following format: **"Wasted litters of water: {wastedLittersOfWater}.**

### Constraints

* All of the given numbers will be valid integers in the range [1, 500].
* It is safe to assume that there will be **NO** case in which the water is **exactly as much** as the cups' values, so that at the end there are no cups and no water in the bottles.
* Allowed time/memory: 100ms/16MB.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 4 2 10 5  3 15 15 11 6 | **Bottles: 3**  **Wasted litters of water: 26** | **We take the first entered cup and the last entered bottle, as it is described in the condition.**  **6 – 4 = 2 – we have 2 more so the wasted water becomes 2.**  **11 – 2 = 9 –** again, it is more, so we add it to the previous amount, which is 2 and it becomes 11.  **15 – 10 = 5 – wasted water becomes 16.**  **15 – 5 = 10 – wasted water becomes 26.**  **We've managed to fill up all of the cups, so we print the remaining bottles and the total amount of wasted water.** |
| **1 5 28 1 4**  **3 18 1 9 30 4 5** | **Cups: 4**  **Wasted litters of water: 35** |  |
| **10 20 30 40 50**  **20 11** | **Cups: 30 40 50**  **Wasted litters of water: 1** |  |