# Final Exam Retake 10.04.2020

## 1. Secret Chat

*You have plenty of free time, so you decide to write a program that conceals and reveals your received messages. Go ahead and type it in!*

On the first line of the input you will receive the **concealed message**. After that, until the "Reveal" command is given, **you will be receiving strings** with **instructions** for different **operations** that need to be performed upon the **concealed message** in order to **interpret** **it** and reveal its true content. There are several types of instructions, split by ":|:"

* InsertSpace:|:{index}
  + Inserts a single **empty space** **at the given index**. The given index will always be valid.
* Reverse:|:{substring}
  + If the message contains the given **substring**, **cut it out**, **reverse** it and **add** it at the **end** of the message.
  + If not, print "error".
  + This operation should replace only the first occurrence of the given **substring** **if there are more than one such occurrences**.
* ChangeAll:|:{substring}:|:{replacement}
  + Changes all occurrences of the given substring with the replacement text.

### Input / Constraints

* On the first line, you will receive a string with message.
* On the next lines, you will be receiving commands, split by **":|:"**.

### Output

* After each set of instructions, print the resulting string.
* After the "Reveal" command is received, print this message:  
  "**You have a new text message: {message}**"

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| heVVodar!gniV  ChangeAll:|:V:|:l  Reverse:|:!gnil  InsertSpace:|:5  Reveal | hellodar!gnil  hellodarling!  hello darling!  You have a new text message: hello darling! |
| **Comments** | |
| **ChangeAll:|:V:|:l** heVVodar!gniV -> hellodar!gnil (We replace all occurrences of "V" with "l")  **Reverse:|:!gnil**  hellodar!gnil -> !gnil -> ling! -> hellodarling! (We reverse !gnil to ling! And put it in the end of the string)  **InsertSpace:|:5**  hellodarling! -> hello.darling! (We insert a space at index 5)  Finally, after receiving the **"Reveal"** command, we print the resulting message. | |
| **Input** | **Output** |
| Hiware?uiy  ChangeAll:|:i:|:o  Reverse:|:?uoy  Reverse:|:jd  InsertSpace:|:3  InsertSpace:|:7  Reveal | Howare?uoy  Howareyou?  error  How areyou?  How are you?  You have a new text message: How are you? |

|  |  |
| --- | --- |
| **JavaScript Input** | **Output** |
| [  'heVVodar!gniV',  'ChangeAll:|:V:|:l',  'Reverse:|:!gnil',  'InsertSpace:|:5',  'Reveal'  ] | hellodar!gnil  hellodarling!  hello darling!  You have a new text message: hello darling! |
| **Comments** | |
| **ChangeAll:|:V:|:l** heVVodar!gniV -> hellodar!gnil (We replace all occurrences of "V" with "l")  **Reverse:|:!gnil**  hellodar!gnil -> !gnil -> ling! -> hellodarling! (We reverse !gnil to ling! And put it in the end of the string)  **InsertSpace:|:5**  hellodarling! -> hello.darling! (We insert a space at index 5)  Finally, after receiving the **"Reveal"** command, we print the resulting message. | |
| **JavaScript Input** | **Output** |
| [  'Hiware?uiy',  'ChangeAll:|:i:|:o',  'Reverse:|:?uoy',  'Reverse:|:jd',  'InsertSpace:|:3',  'InsertSpace:|:7',  'Reveal'  ] | Howare?uoy  Howareyou?  error  How areyou?  How are you?  You have a new text message: How are you? |

## 2. Mirror words

*The SoftUni Spelling Bee competition is here. But it`s not like any other Spelling Bee competition out there, it`s different and a lot more fun! You, of course, are a participant and you are eager to show the competition that you are the best, so go ahead, learn the rules and win!*

On the first line of the input you will be given a **text string**. In order to win the competition you have to find all hidden **word pairs**, read them and mark the ones that are **mirror** **images** of each other.

First of all you have to **extract the hidden word pairs**. Hidden word pairs are:

* Surrounded by "@" or "#" (only one of the two) in the following pattern #wordOne##wordTwo# or @wordOne@@wordTwo@
* At least **3 characters long each** (**without the surrounding symbols**)
* Made up of **letters** **only**

If the second word, **spelled backwards** is the **same** **as the first word** **and vice versa** (**casing matters**!), then they are a **match** and you have to store them somewhere. **Examples** of mirror words:

#Part##traP# @leveL@@Level@ #sAw##wAs#

* If you don`t find any valid pairs print: **"No word pairs found!"**
* If you find valid pairs print their count: **"{valid pairs count} word pairs found!"**
* If there are no mirror words print: **"No mirror words!"**
* If there are mirror words print:

"The mirror words are:

{wordOne} <=> {wordtwo}, {wordOne} <=> {wordtwo}, {wordOne} <=> {wordtwo}, etc…"

### Input / Constraints

* You will recive a string.

### Output

* Print the proper output messages in the proper cases as described in the problem description.
* If there are pairs of mirror words, print them in the end, each pair separated by **", "**.
* Each pair of mirror word must be printed with **" <=> "** between the words.

### Examples

|  |  |
| --- | --- |
| **Input** | |
| @mix#tix3dj#poOl##loOp#wl@@bong&song%4very$long@thong#Part##traP##@@leveL@@Level@##car#rac##tu@pack@@ckap@#rr#sAw##wAs#r#@w1r | |
| **Output** | **Comments** |
| 5 word pairs found!  The mirror words are:  Part <=> traP, leveL <=> Level, sAw <=> wAs | There are 5 green and yellow pairs that meet all requirements and thus are valid.  #poOl##loOp# is valid and looks very much like a mirror words pair but it isn`t because the casings don`t match.  #car#rac# “rac” spelled backwards is "car" but this is not a valid pair because there is only one "#" between the words.  @pack@@ckap@ is also valid but "ckap" backwards is "pakc" which is not the same as "pack", so they are not mirror words. |
| **Input** | |
| #po0l##l0op# @bAc##cAB@ @LM@ML@ #xxxXxx##xxxXxx# @aba@@ababa@ | |
| **Output** | **Comments** |
| 2 word pairs found!  No mirror words! | "xxxXxx" backwards is not the same as "xxxXxx"  @aba@@ababa@ is a valid pair but the word lengths are different, thus these are definitely not mirror words |
| **Input** | |
| #lol#lol# @#God@@doG@# #abC@@Cba# @Xyu@#uyX# | |
| **Output** | **Comments** |
| No word pairs found!  No mirror words! |  |

|  |  |
| --- | --- |
| **JavaScript Input** | |
| [  '@mix#tix3dj#poOl##loOp#wl@@bong&song%4very$long@thong#Part##traP##@@leveL@@Level@##car#rac##tu@pack@@ckap@#rr#sAw##wAs#r#@w1r'  ] | |
| **Output** | **Comments** |
| 5 word pairs found!  The mirror words are:  Part <=> traP, leveL <=> Level, sAw <=> wAs | There are 5 green and yellow pairs that meet all requirements and thus are valid.  #poOl##loOp# is valid and looks very much like a mirror words pair but it isn`t because the casings don`t match.  #car#rac# “rac” spelled backwards is "car" but this is not a valid pair because there is only one "#" between the words.  @pack@@ckap@ is also valid but "ckap" backwards is "pakc" which is not the same as "pack", so they are not mirror words. |
| **JavaScript Input** | |
| [ '#po0l##l0op# @bAc##cAB@ @LM@ML@ #xxxXxx##xxxXxx# @aba@@ababa@' ] | |
| **Output** | **Comments** |
| 2 word pairs found!  No mirror words! | "xxxXxx" backwards is not the same as "xxxXxx"  @aba@@ababa@ is a valid pair but the word lengths are different, thus these are definitely not mirror words |
| **JavaScript Input** | |
| [ '#lol#lol# @#God@@doG@# #abC@@Cba# @Xyu@#uyX#' ] | |
| **Output** | **Comments** |
| No word pairs found!  No mirror words! |  |

## 3. Need for Speed III

*You have just bought the latest and greatest computer game – Need for Seed III. We know that you can`t wait to start playing. Pick your favorite cars and drive them all you want!*

On the first line of the standard input you will receive an integer **n** – the **number of cars** that you can obtain. On the next **n** lines the **cars themselves** will follow with their **mileage** and **fuel** **available**, separated by "|" in the following format:

{car}|{mileage}|{fuel}

Then, you will be receiving different **commands**, each on a new line, separated by " : ", until the "Stop" command is given:

* Drive : {car} : {distance} : {fuel}
  + You need to **drive the given distance** and you will **need the given** fuel to do that. If the car **doesn`t have enough fuel** print:  
    "**Not enough fuel to make that ride**"
  + If the car has the required fuel available in the tank, **increase its mileage** with **the given distance**, **decrease its fuel with the given fuel** and **print**:   
    "{car} driven for {distance} kilometers. {fuel} liters of fuel consumed."
  + You like driving new cars only, so if the mileage of a car reaches **100 000** km, remove it from the collection(s). Print:  
    "**Time to sell the {car}!**"
* Refuel : {car} : {fuel}
  + Refill the tank of your car.
  + Each tank can hold a **maximum of 75 liters of fuel**, so if the given amount of fuel is more than you can fit in the tank, take only what is required to fill it up.
  + Print a message in the following format:  
    "{car} refueled with {fuel} liters"
* Revert : {car} : {kilometers}
  + Decrease the **mileage** of the given **car with the given kilometers** and print the kilometers you have decreased it with in the following format:  
    "{car} mileage decreased by {amount reverted} kilometers"
  + If the mileage becomes **less** **than** **10 000km** **after** it is decreased, **just set it to 10 000km** and   
    **DO NOT print anything.**

Upon receiving the "Stop" command you need to print all cars in your possession, sorted by their **mileage in decscending order**, then by their **name in ascending order**, in the following format:  
"**{car} -> Mileage: {mileage} kms, Fuel in the tank: {fuel} lt.**"

### Input/Constraints

* The **mileage** and **fuel** of the cars will be valid, 32-bit integers and will never be negative.
* The **fuel** and **distance** amounts **in the commands will never be negative**.
* The **car** **names** in the **commands** will always be **valid cars in your possession**.

### Output

* All the output messages with the appropriate formats are described in the problem description.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  Audi A6|38000|62  Mercedes CLS|11000|35  Volkswagen Passat CC|45678|5  Drive : Audi A6 : 543 : 47  Drive : Mercedes CLS : 94 : 11  Drive : Volkswagen Passat CC : 69 : 8  Refuel : Audi A6 : 50  Revert : Mercedes CLS : 500  Revert : Audi A6 : 30000  Stop | Audi A6 driven for 543 kilometers. 47 liters of fuel consumed.  Mercedes CLS driven for 94 kilometers. 11 liters of fuel consumed.  Not enough fuel to make that ride  Audi A6 refueled with 50 liters  Mercedes CLS mileage decreased by 500 kilometers  Volkswagen Passat CC -> Mileage: 45678 kms, Fuel in the tank: 5 lt.  Mercedes CLS -> Mileage: 10594 kms, Fuel in the tank: 24 lt.  Audi A6 -> Mileage: 10000 kms, Fuel in the tank: 65 lt. |
| **Comments** | |
| After we receive the cars with their mileage and fuel, we start driving them. When we get to "**Drive : Volkswagen Passat CC : 69 : 8**" command, our program calculates that there is not enough fuel and we print the appropriate message. Then we refuel the Audi A6 with 50 l of fuel and Revert the Mercedes with 500 kilometers.  When we receive the "Revert : Audi A6 : 30000", we set its mileage to **10000** km, because if the current mileage of the Audi is **38543** kms and if we subtract **30000** from it, we receive **8543** kms, which is less than 10000 kms.  After all the commands, we print our current collection of cars with their current mileage and current fuel. | |
| **Input** | **Output** |
| 4  Lamborghini Veneno|11111|74  Bugatti Veyron|12345|67  Koenigsegg CCXR|67890|12  Aston Martin Valkryie|99900|50  Drive : Koenigsegg CCXR : 382 : 82  Drive : Aston Martin Valkryie : 99 : 23  Drive : Aston Martin Valkryie : 2 : 1  Refuel : Lamborghini Veneno : 40  Revert : Bugatti Veyron : 2000  Stop | Not enough fuel to make that ride  Aston Martin Valkryie driven for 99 kilometers. 23 liters of fuel consumed.  Aston Martin Valkryie driven for 2 kilometers. 1 liters of fuel consumed.  Time to sell the Aston Martin Valkryie!  Lamborghini Veneno refueled with 1 liters  Bugatti Veyron mileage decreased by 2000 kilometers  Koenigsegg CCXR -> Mileage: 67890 kms, Fuel in the tank: 12 lt.  Lamborghini Veneno -> Mileage: 11111 kms, Fuel in the tank: 75 lt.  Bugatti Veyron -> Mileage: 10345 kms, Fuel in the tank: 67 lt. |

|  |  |
| --- | --- |
| **JavaScript Input** | **Output** |
| [  '3',  'Audi A6|38000|62',  'Mercedes CLS|11000|35',  'Volkswagen Passat CC|45678|5',  'Drive : Audi A6 : 543 : 47',  'Drive : Mercedes CLS : 94 : 11',  'Drive : Volkswagen Passat CC : 69 : 8',  'Refuel : Audi A6 : 50',  'Revert : Mercedes CLS : 500',  'Revert : Audi A6 : 30000',  'Stop'  ] | Audi A6 driven for 543 kilometers. 47 liters of fuel consumed.  Mercedes CLS driven for 94 kilometers. 11 liters of fuel consumed.  Not enough fuel to make that ride  Audi A6 refueled with 50 liters  Mercedes CLS mileage decreased by 500 kilometers  Volkswagen Passat CC -> Mileage: 45678 kms, Fuel in the tank: 5 lt.  Mercedes CLS -> Mileage: 10594 kms, Fuel in the tank: 24 lt.  Audi A6 -> Mileage: 10000 kms, Fuel in the tank: 65 lt. |
| **Comments** | |
| After we receive the cars with their mileage and fuel, we start driving them. When we get to "**Drive : Volkswagen Passat CC : 69 : 8**" command, our program calculates that there is not enough fuel and we print the appropriate message. Then we refuel the Audi A6 with 50 l of fuel and Revert the Mercedes with 500 kilometers.  When we receive the "Revert : Audi A6 : 30000", we set its mileage to **10000** km, because if the current mileage of the Audi is **38543** kms and if we subtract **30000** from it, we receive **8543** kms, which is less than 10000 kms.  After all the commands, we print our current collection of cars with their current mileage and current fuel. | |
| **JavaScript Input** | **Output** |
| [  '4',  'Lamborghini Veneno|11111|74',  'Bugatti Veyron|12345|67',  'Koenigsegg CCXR|67890|12',  'Aston Martin Valkryie|99900|50',  'Drive : Koenigsegg CCXR : 382 : 82',  'Drive : Aston Martin Valkryie : 99 : 23',  'Drive : Aston Martin Valkryie : 2 : 1',  'Refuel : Lamborghini Veneno : 40',  'Revert : Bugatti Veyron : 2000',  'Stop'  ] | Not enough fuel to make that ride  Aston Martin Valkryie driven for 99 kilometers. 23 liters of fuel consumed.  Aston Martin Valkryie driven for 2 kilometers. 1 liters of fuel consumed.  Time to sell the Aston Martin Valkryie!  Lamborghini Veneno refueled with 1 liters  Bugatti Veyron mileage decreased by 2000 kilometers  Koenigsegg CCXR -> Mileage: 67890 kms, Fuel in the tank: 12 lt.  Lamborghini Veneno -> Mileage: 11111 kms, Fuel in the tank: 75 lt.  Bugatti Veyron -> Mileage: 10345 kms, Fuel in the tank: 67 lt. |

# Final Exam 04.04.2020

## 1. Password Reset

*Yet again you have forgotten your password... Naturally it`s not the first time this has happened. Actually you got so tired of it that you decided to help yourself with a smart solution.*

Write a password reset program that performs a series of commands upon a predefined string. First, you will receive a string and afterwards, until the command "**Done**" is given, you will be receiving strings with commands split by a single space. The commands will be the following:

* TakeOdd
  + Takes only the characters at **odd** **indices** and **concatenates** them together to  
    obtain the **new raw password** and then **prints** it.
* Cut {index} {length}
  + Gets the substring with the **given length** starting from the **given index** from the password and removes its first occurrence of it, then prints the password on the console.
  + The given index and length will **always** be **valid**.
* Substitute {substring} {substitute}
  + If the raw password contains the given substring, replaces all of its   
    occurrences with the substitute text given and prints the result.
  + If it doesn’t, prints "Nothing to replace!"

### Input

* You will be receiving strings until the "**Done**" command is given.

### Output

* After the "Done" command is received, print:
  + "Your password is: {password}"

### Constraints

* The indexes from the "**Cut {index} {length}**" command will always be valid.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Siiceercaroetavm!:?:ahsott.:i:nstupmomceqr  TakeOdd  Cut 15 3  Substitute :: -  Substitute | ^  Done | icecream::hot::summer  icecream::hot::mer  icecream-hot-mer  Nothing to replace!  Your password is: icecream-hot-mer |
| **Comments** | |
| TakeOdd  Siiceercaroetavm!:?:ahsott.:i:nstupmomceqr -> icecream::hot::summer  We only take the chars at odd indices 1, 3, 5 etc.  **Cut 15 3 ->** icecream::hot::summer -> sum  icecream::hot::mer  We cut a substring starting at index 15 with length 3,  remove it from the raw password and print it.  Then, on a new line we print the resulting new raw password.  **Substitute :: - ->** icecream::hot::summer -> icream-hot-summer  We replace "::" with "-".  **Substitute** | ^ **->** Nothing to replace!  "|" is not found anywhere in the raw password.  Finally, after receiving the "**Done**" command, we print the resulting password in the proper format. | |
| **Input** | **Output** |
| up8rgoyg3r1atmlmpiunagt!-irs7!1fgulnnnqy  TakeOdd  Cut 18 2  Substitute ! \*\*\*  Substitute ? .!.  Done | programming!is!funny  programming!is!fun  programming\*\*\*is\*\*\*fun  Nothing to replace!  Your password is: programming\*\*\*is\*\*\*fun |

## 2. Fancy Barcodes

Your first task is to determine if the given sequence of characters is a **valid** barcode or **not**.

**Each line must not contain anything else but a valid barcode**. A barcode is **valid** when:

* Is surrounded with a "@" followed by one or more "#"
* Is **at least 6 characters long** (without the surrounding "@" or "#")
* **Starts** with a **capital letter**
* Contains **only letters** (lower and upper case) **and digits**
* **Ends** with a **capital letter**

Examples of valid barcodes: @#FreshFisH@#, @###Brea0D@###, @##Che46sE@##, @##Che46sE@###

Examples of invalid barcodes: **##InvaliDiteM##**, **@InvalidIteM@**, **@#Invalid\_IteM@#**

Next you have to determine the **product group** of the item from the **barcode**. The product group is obtained by **concatenating** **all the digits** found in the barcode. If there are **no digits** present in the barcode, the **default** product group is "00".

Examples:

@#FreshFisH@# -> product group: 00

@###Brea0D@### -> product group: 0

@##Che4s6E@## -> product group: 46

### Input

On the first line you will be given an integer **n** – the count of barcodes that you will be receiving next.

On the next **n** lines, you will receive different strings.

### Output

For each barcode that you process, you need to print a message.

If the barcode is invalid:

* "Invalid barcode"

If the barcode is valid:

* "Product group: {product group}"

### Constraints

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  @#FreshFisH@#  @###Brea0D@###  @##Che4s6E@## | Product group: 00  Product group: 0  Product group: 46 |
| **Input** | **Output** |
| 6  @###Val1d1teM@###  @#ValidIteM@#  ##InvaliDiteM##  @InvalidIteM@  @#Invalid\_IteM@#  @#ValiditeM@# | Product group: 11  Product group: 00  Invalid barcode  Invalid barcode  Invalid barcode  Product group: 00 |

## 3. Heroes of Code and Logic VII

*You got your hands on the most recent update on the best MMORPG of all time – Heroes of Code and Logic. You want to play it all day long! So cancel all other arrangements and create your party!*

On the first line of the standard input you will receive an integer **n** – the number of heroes that you can choose for your party. On the next **n** lines, the heroes themselves will follow with their **hit points** and **mana points** separated by empty space in the following format:

{hero name} {HP} {MP}

* where HP stands for hit points and MP for mana points
* a hero can have a maximum of 100 HP and 200 MP

After you have successfully picked your heroes, you can start playing the game. You will be receiving different commands, each on a new line, separated by " – ", until the "End" command is given.

There are several actions that can be performed by the heroes:

CastSpell – {hero name} – {MP needed} – {spell name}

* If the hero has the required MP, he casts the spell, thus reducing his MP. Print this message:
  + "{hero name} has successfully cast {spell name} and now has {mana points left} MP!"
* If the hero is unable to cast the spell print:
  + "**{hero name} does not have enough MP to cast {spell name}!**"

TakeDamage – {hero name} – {damage} – {attacker}

* Reduce the hero HP by the given damage amount. If the hero is still alive (his HP is greater than 0) print:
  + "{hero name} was hit for {damage} HP by {attacker} and now has {current HP} HP left!"
* If the hero has died, remove him from your party and print:
  + "{hero name} has been killed by {attacker}!"

Recharge – {hero name} – {amount}

* The hero increases his MP. If a command is given that would bring the MP of the hero above **200**, MP is increased so that it reaches the **maximum**. Print the following message:
  + "{hero name} recharged for {amount recovered} MP!"

Heal – {hero name} – {amount}

* The hero increases his HP. If a command is given that would bring the HP of the hero above **100**, HP is increased so that it reaches the **maximum**. Print the following message:
  + "{hero name} healed for {amount recovered} HP!"

### Input

* On the first line of the standard input you will receive an integer **n**
* On the next **n** lines, the heroes themselves will follow with their **hit points** and **mana points** separated by empty space in the following format
* You will be receiving different **commands**, each on a new line, separated by " – ", until the "End" command is given

### Output

* Print all members of your party who are **still alive**, sorted by their **HP in descending order**, then by their **name in ascending order**, in the following format (their HP/MP need to be indented 2 spaces):

"{hero name}

HP: {current HP}

MP: {current MP}

..."

### Constraints

* The starting HP/MP of the heroes will be valid, 32-bit integers, will never be negative or exceed the respective limits.
* The HP/MP amounts in the commands will never be negative.
* The hero names in the commands will always be valid members of your party. No need to check that explicitly

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  Solmyr 85 120  Kyrre 99 50  Heal - Solmyr - 10  Recharge - Solmyr - 50  TakeDamage - Kyrre - 66 - Orc  CastSpell - Kyrre - 15 - ViewEarth  End | Solmyr healed for 10 HP!  Solmyr recharged for 50 MP!  Kyrre was hit for 66 HP by Orc and now has 33 HP left!  Kyrre has successfully cast ViewEarth and now has 35 MP!  Solmyr  HP: 95  MP: 170  Kyrre  HP: 33  MP: 35 |
| **Comments** | | |
| These are examples of successful actions. The different colors denote the commands and their respective messages. | | |
| **Input** | **Output** |
| 4  Adela 90 150  SirMullich 70 40  Ivor 1 111  Tyris 94 61  Heal - SirMullich - 50  Recharge - Adela - 100  CastSpell - Tyris - 1000 - Fireball  TakeDamage - Tyris - 99 - Fireball  TakeDamage - Ivor - 3 - Mosquito  End | SirMullich healed for 30 HP!  Adela recharged for 50 MP!  Tyris does not have enough MP to cast Fireball!  Tyris has been killed by Fireball!  Ivor has been killed by Mosquito!  SirMullich  HP: 100  MP: 40  Adela  HP: 90  MP: 200 |
| **Comments** | | |
| Heal – SirMullich healed for 30 HP due to the HP max limit.  Recharge – Adela recharged for 50 MP due to the MP max limit.  CastSpell – Tyris does not have enough MP to cast the spell.  TakeDamage – Tyris`s HP is reduced by 99, thus becoming -5, which means that he is dead.  TakeDamage – Ivor`s HP is now -2, so he is dead too.  After the "End" command we print the remaining living heroes, sorted by their HP in reverse order. | | |

# Final Exam 04.04.2020

## 1. Activation Keys

*You are about to make some good money, but first you need to think of a way to verify who paid for your product and who didn`t. You have decided to let people use the software for a free trial period and then require an activation key in order to continue to use the product. The last step before you could cash out is to design a program that creates unique activation keys for each user. So, waste no more time and start typing!*

The first line of the input will be your raw activation key. It will consist of **letters and numbers only**.

After that, until the "Generate" command is given, you will be receiving strings with instructions for different operations that need to be performed upon the raw activation key.

There are several types of instructions, split by ">>>":

* Contains>>>{substring} – checks if the raw activation key contains the given substring.
  + If it does prints: "{raw activation key} contains {substring}".
  + If not, prints: "Substring not found!"
* Flip>>>Upper/Lower>>>{startIndex}>>>{endIndex}
  + Changes the substring **between the given indices (the end index is exclusive)** to upper or lower case.
  + All given indexes will be valid.
  + Prints the activation key.
* **Slice>>>{startIndex}>>>{endIndex}**
  + **Deletes** the characters between the start and end indices (**end index is exclusive**).
  + Both indices will be **valid**.
  + Prints the activation key.

### Input

* The first line of the input will be string and it will consist of **letters and numbers only**.
* After the first line, until the "Generate" command is given, you will be receiving **strings**.

### Output

* After the "Generate" command is received, print:
  + "Your activation key is: {activation key}"

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| abcdefghijklmnopqrstuvwxyz  Slice>>>2>>>6  Flip>>>Upper>>>3>>>14  Flip>>>Lower>>>5>>>7  Contains>>>def  Contains>>>deF  Generate | abghijklmnopqrstuvwxyz  abgHIJKLMNOPQRstuvwxyz  abgHIjkLMNOPQRstuvwxyz  Substring not found!  Substring not found!  Your activation key is: abgHIjkLMNOPQRstuvwxyz |
| **Comments** | |
| 1. **Slice>>2>>6**   abcdefghijklmnopqrstuvwxyz becomes abghijklmnopqrstuvwxyz   1. **Flip>>>Upper>>>3>>>14**   abghijklmnopqrstuvwxyz becomes abgHIJKLMNOPQRstuvwxyz   1. **Flip>>>Lower>>>5>>>7**   abgHIJKLMNOPQRstuvwxyz becomes abgHIjkLMNOPQRstuvwxyz   1. **Contains>>>def**   abgHIjkLMNOPQRstuvwxyz does not contain def   1. **Contains>>>deF**   abgHIjkLMNOPQRstuvwxyz does not contain deF  The final activation key is abgHIjkLMNOPQRstuvwxyz | |
| **Input** | **Output** |
| 134softsf5ftuni2020rockz42  Slice>>>3>>>7  Contains>>>-rock  Contains>>>-uni-  Contains>>>-rocks  Flip>>>Upper>>>2>>>8  Flip>>>Lower>>>5>>>11  Generate | 134sf5ftuni2020rockz42  Substring not found!  Substring not found!  Substring not found!  134SF5FTuni2020rockz42  134SF5ftuni2020rockz42  Your activation key is: 134SF5ftuni2020rockz42 |

## 2. Emoji Detector

Your task is to write program which extracts emojis from a text and find the threshold based on the input.

You have to get your **cool threshold**. It is obtained by **multiplying all** the digits found in the input. The cool threshold could be a **very big number**, so be mindful.

An emoji is valid when:

* Is surrounded by either :: or \*\* (exactly 2)
* Is **at least 3** characters long (**without** the surrounding symbols)
* **Starts** with a **capital letter**
* Continues with **lowercase** letters **only**

Examples of valid emojis: ::Joy::, \*\*Banana\*\*, ::Wink::

Examples of invalid emojis: ::Joy\*\*, **::fox:es:**, **\*\*Monk3ys\*\*, :Snak::Es::**

You need to count **all valid emojis** in the text and calculate their **coolness**. The coolness of the emoji is **determined** by summing all the **ASCII values of all letters** in the emoji.

Examples: ::Joy:: - 306, \*\*Banana\*\* - 577, ::Wink:: - 409

You need to print the result of cool threshold and after that to take all emojis out of the text, count them and print the **only the cool ones** on the console.

### Input

* On the single input you will receive a piece of string.

### Output

* On the first line of the output print the obtained Cool threshold in format:
* **Cool threshold: {coolThresholdSum}**

On the next line **print the** **count of all emojis** found in the text in format:

* {countOfAllEmojis} emojis found in the text. The cool ones are:
* {cool emoji 1}
* {cool emoji 2}
* {…}

If there are no cool ones, just don't print anything in the end.

### Constraints

There will always be at least one digit in the text!

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| In the Sofia Zoo there are 311 animals in total! ::Smiley:: This includes 3 \*\*Tigers\*\*, 1 ::Elephant:, 12 \*\*Monk3ys\*\*, a \*\*Gorilla::, 5 ::fox:es: and 21 different types of :Snak::Es::. ::Mooning:: \*\*Shy\*\* | Cool threshold: 540  4 emojis found in the text. The cool ones are:  ::Smiley::  \*\*Tigers\*\*  ::Mooning:: |
| **Comments** | |
| You can see all the valid emojis in green. There are various reasons why the rest are not valid, examine them carefully. The "cool threshold" is 3\*1\*1\*3\*1\*1\*2\*3\*5\*2\*1 = 540.  ::Smiley:: -> 83 + 109 + 105 + 108 + 101 + 121 = 627 > 540 -> cool  \*\*Tigers\*\* -> 84 + 105 + 103 + 101 + 114 + 115 = 622 > 540 -> cool  ::Mooning:: -> 77 + 111 + 111 + 112 + 105 + 112 + 103 = 727 > 540 -> cool  \*\*Shy\*\* -> 83 + 104 + 121 = 308 < 540 -> not cool  At the end we print the count of all valid emojis found and each of the cool ones on a new line. | |
| **Input** | **Output** |
| 5, 4, 3, 2, 1, go! The 1-th consecutive banana-eating contest has begun! ::Joy:: \*\*Banana\*\* ::Wink:: \*\*Vali\*\* ::valid\_emoji:: | Cool threshold: 120  4 emojis found in the text. The cool ones are:  ::Joy::  \*\*Banana\*\*  ::Wink::  \*\*Vali\*\* |
| **Input** | **Output** |
| It is a long established fact that 1 a reader will be distracted by 9 the readable content of a page when looking at its layout. The point of using ::LoremIpsum:: is that it has a more-or-less normal 3 distribution of 8 letters, as opposed to using 'Content here, content 99 here', making it look like readable \*\*English\*\*. | Cool threshold: 17496  1 emojis found in the text. The cool ones are: |
| **Comments** | |
| You can see \*\*English\*\* is a valid emoji, but the sum of ascii **is not** **bigger** than cool threshold, that's why we **don't** print anything in the end. | |

## 3. P!rates

*Anno 1681. The Caribbean. The golden age of piracy. You are a well-known pirate captain by the name of Jack… Daniels. Together with your comrades Jim (Beam) and Johnny (Walker) you have been roaming the seas, looking for gold and treasure… and the occasional killing, of course. Go ahead, target some wealthy settlements and show them the pirate`s way!*

### Description

Until the "Sail" command is given you will be receiving:

* Cities that you and your crew have targeted, with their **population** and **gold**, separated by "||".
* If you receive a city which has been already received, you have to increase the population and gold with the given values.

After the "Sail" command, you will start receiving lines of text representing events until the "End" command is given.

Events will be in the following format:

* "Plunder=>{town}=>{people}=>{gold}"
  + You have successfully attacked and plundered the town, killing the given number of people and stealing the respective amount of gold.
  + For every town you attack print this message: "{town} plundered! {gold} gold stolen, {people} citizens killed."
  + If any of those two values (population or gold) **reaches zero**, the town is disbanded.
    - You need to **remove it** from your collection of targeted cities and print the following message: "{town} has been wiped off the map!"
  + There will be no case of receiving more people or gold than there is in the city.
* "Prosper=>{town}=>{gold}"
  + There has been a dramatic economic growth in the given city**, increasing its treasury** by the given amount of gold.
  + The gold amount **can be a negative number, so be careful.** If a negative amount of gold is given print: "Gold added cannot be a negative number!" and ignore the command.
  + If the given gold is a valid amount, increase the town's gold reserves by the respective amount and print the following message: "{gold added} gold added to the city treasury. {town} now has {total gold} gold."

### Input

* On the first lines, until the **"Sail"** command, you will be receiving strings representing the cities with their gold and population, separated by **"||"**
* On the next lines, until the **"End"** command, you will be receiving strings representing the actions described above, separated by **"=>"**

### Output

* After receiving the "End" command if there are any existing settlements on your list of targets, you need to print all of them, sorted by their **gold in descending order**, then by their **name in ascending order**, in the following format:

Ahoy, Captain! There are {count} wealthy settlements to go to:

{town1} -> Population: {people} citizens, Gold: {gold} kg

…

{town…n} -> Population: {people} citizens, Gold: {gold} kg

* If there are no settlements left to plunder, print:

"Ahoy, Captain! All targets have been plundered and destroyed!"

### Constraints

* The initial population and gold of the settlements will be valid, 32-bit integers,   
  will never be negative or exceed the respective limits.
* The town names in the events will always be valid towns that should be on your list.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Tortuga||345000||1250  Santo Domingo||240000||630  Havana||410000||1100  Sail  Plunder=>Tortuga=>75000=>380  Prosper=>Santo Domingo=>180  End | Tortuga plundered! 380 gold stolen, 75000 citizens killed.  180 gold added to the city treasury. Santo Domingo now has 810 gold.  Ahoy, Captain! There are 3 wealthy settlements to go to:  Havana -> Population: 410000 citizens, Gold: 1100 kg  Tortuga -> Population: 270000 citizens, Gold: 870 kg  Santo Domingo -> Population: 240000 citizens, Gold: 810 kg |
| **Input** | **Output** |
| Nassau||95000||1000  San Juan||930000||1250  Campeche||270000||690  Port Royal||320000||1000  Port Royal||100000||2000  Sail  Prosper=>Port Royal=>-200  Plunder=>Nassau=>94000=>750  Plunder=>Nassau=>1000=>150  Plunder=>Campeche=>150000=>690  End | Gold added cannot be a negative number!  Nassau plundered! 750 gold stolen, 94000 citizens killed.  Nassau plundered! 150 gold stolen, 1000 citizens killed.  Nassau has been wiped off the map!  Campeche plundered! 690 gold stolen, 150000 citizens killed.  Campeche has been wiped off the map!  Ahoy, Captain! There are 2 wealthy settlements to go to:  Port Royal -> Population: 420000 citizens, Gold: 3000 kg  San Juan -> Population: 930000 citizens, Gold: 1250 kg |

# Final Exam 13 December 2019

## Warrior’s Quest

*Warrior! Thank you for coming. The city is overrun by demons. We need your help securing the civilians and defending our land. For the Alliance!*

First, you will receive **a skill that needs the deciphered.**

Next, you will be receiving **commands** split by a single space until you get the "**For Azeroth**" command. There are **5 possible commands:**

* **"GladiatorStance"**
  + **Replace** all letters with **upper case** and **print** the result.
* **"DefensiveStance"**
  + **Replace** all letters with **lower case** and **print** the result.
* **"Dispel {index} {letter}"**
  + **Replace** the letter at the index with the given one and print **"Success!"**
  + If the index is **invalid,** print: **"Dispel too weak."**
* **"Target Change {substring} {second substring}"**
  + **Replace** the **first substring** with the **second** and **print the result**.
* **"Target Remove {substring}"**
  + **Remove** the **substring from the string** and **print the result**.

If the input command is **not** in the list, print **"Command doesn't exist!"**

### Input / Constraints

* On the **1st line** you are going to receive the **string**.
* On the next **lines**, until you receive **"For Azeroth"**, you will be receiving commands.
* All commands are **case sensitive**.

### Output

* **Print** the **output** of the **commands** in the **format described above**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| fr1c710n  GladiatorStance  Dispel 2 I  Dispel 4 T  Dispel 6 O  Dispel 5 I  Dispel 10 I  Target Change RICTION riction  For Azeroth | FR1C710N  Success!  Success!  Success!  Success!  Dispel too weak.  Friction |
| **Input** | **Output** |
| DYN4MICNIC  Target Remove NIC  Dispel 3 A  DefensiveStance  Target Change d D  target change D d  For Azeroth | DYN4MIC  Success!  dynamic  Dynamic  Command doesn't exist! |

## Boss Rush

*Ah, look who arrived, the latecomer. Let me repeat our plan for our friend here. There is a hoard of demons in this room. We must first identify them before we can proceed.*

Create a program that **checks** if **inputs** are **valid** and **decrypt it**. On the **first** line you will **receive** a **number** that **indicates** how **many inputs** you will **receive** on the **next** lines.

You will read lines with a boss name and title and you should check if they are valid, considering the following rules:

* **Boss** - the name should be in **upper case letters**, should be minimum **four letters long** and should be surrounded by **"|"**
* **Title** - contains **exactly 2 words** and they **contain** only alphabetical letters and **1 whitespace** between them. The title should be surrounded by **"#"**
* The name and title **should be split by a single ":"**

**Example for a valid input:** |GEORGI|:#Lead architect#

If the input is valid. Print in the following format:

**"{boss name}, The {title}**

**>> Strength: {length of the name}**

**>> Armour: {length of the title}"**

If the input is invalid, print **"Access denied!"**

### Input / Constraints

* On the **1st line**, you will receive a **number** of inputs.
* On the next **n lines**, you will have to **check** if a boss has a **valid name and title**.

### Output

* **Print** the **output** with the **format described above.**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  |GEORGI|:#Lead architect#  |Hristo|:#High Overseer#  |STEFAN|:#Assistant Game Developer# | GEORGI, The Lead architect  >> Strength: 6  >> Armour: 14  Access denied!  Access denied! |
| **Input** | **Output** |
| 3  |PETER|:#H1gh Overseer#  |IVAN|:#Master detective#  |KARL|: #Marketing lead# | Access denied!  IVAN, The Master detective  >> Strength: 4  >> Armour: 16  Access denied! |

## 3. Hero Recruitment

*Greetings, champions of Azeroth. Welcome to the Chapels. Here, we will test your skill and knowledge to decide if you are ready to defend our beloved world. I wish you luck champions. For Honour and Glory!*

Create a program that keeps track of enrolled heroes and their collection of spells (spellbook). You will be receiving the following commands until you receive the command **"End"**:

**"Enroll {HeroName}":**

* Adds the hero to your collection of heroes.
* If the hero is already present in your collection, print: **"{HeroName} is already enrolled."**

**"Learn {HeroName} {SpellName}":**

* Adds the **{SpellName}** to the **{HeroName}**'s spellbook.
* If the **{HeroName}** doesn’t exist in the collection, print: "**{HeroName} doesn't exist."**
* If the hero already has the spell in his spellbook print: **S"**

**"Unlearn {HeroName} {SpellName}":**

* Remove the **{SpellName}** from the **{HeroName}**'s spellbook.
* If the **{HeroName}** doesn’t exist in the collection, print: "**{HeroName} doesn't exist."**
* If the **{SpellName}** doesn't exist in the hero's spellbook, print: **"{HeroName} doesn't know {SpellName}."**

After you receive the **"End"** command, print all the heroes sorted by their count of spells in **descending** and then by the hero name **ascending** in the format described below:

**"Heroes:**

**== {name1}: {spell1}, {spell2}, {spelln}**

**== {name2}: {spell1}, {spell2}, {spelln}**

**...**

**== {nameN}: {spell1}, {spell2}, {spelln}**

### Input / Constraints

* You will be receiving **lines** until you receive the **"End"** command.

### Output

* Print the **heroes** in the **format** described above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Enroll Stefan  Enroll Pesho  Enroll Stefan  Learn Stefan ItShouldWork  Learn Stamat ItShouldNotWork  Unlearn Gosho Dispel  Unlearn Stefan ItShouldWork  End | Stefan is already enrolled.  Stamat doesn't exist.  Gosho doesn't exist.  Heroes:  == Pesho:  == Stefan: |
| **Input** | **Output** |
| Enroll Stefan  Learn Stefan ItShouldWork  Learn Stefan ItShouldWork  Unlearn Stefan NotFound  End | Stefan has already learnt ItShouldWork.  Stefan doesn't know NotFound.  Heroes:  == Stefan: ItShouldWork |

# Final Exam - 07 December 2019 Group 2

## 1. Nikulden’s Charity

*After his name day party, Nikolcho wants to find out how much money he has raised for charity. He received an encrypted message with the amount collected. Your task is to help Nikolcho to figure out how much money he has collected.*

First, you are going to **receive** **a string**, then commands.

You will receive "decrypting" **commands** until you get the "**Finish**" command.

* **"Replace {currentChar} {newChar}"**
  + **Replace** all occurrences of **{currentChar}** with **{newChar},** then **print** the **current message**.
* "**Cut {startIndex} {endIndex}"**
  + **Remove** the substring from the **{startIndex}** until the **{endIndex}**, then **print** the **current message**.
  + If any of the indexes is not valid, **print:**

**"Invalid indexes!"**

* "**Make {Upper/Lower}"**
  + **Replace** all letters with **upper/lowe**r case and **print** the **current message**
* **"Check" {string}**
  + **Check**  if the message contains the given string.
    - If it does, **print:** **"Message contains {string}"**
    - If it doesn’t, **print: "Message doesn't contain {string}"**
* "**Sum {startIndex} {endIndex}"**
  + Get the **substring** from **{startIndex}** to **{endIndex}** and **print** the **sum of the ASCII values of the substring.**
  + If any of the **{startIndex}** or **{endIndex}** are invalid, **print:**

**"Invalid indexes!"**

**Note: At any time, the message will contain at least one character.**

### Input

* On the **1st line** you are going to receive the **string**.
* On the next **lines**, until the **"Finish"** command is received, you will be receiving commands.

### Output

* **Print** the **output** of every **command** in the **format** **described** **above**.

### Constraints

* The **indexes** will be integers in the range [**-50**…**50**]

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| ILikeSharan  Replace a e  Make Upper  Check SHEREN  Sum 1 4  Cut 1 4  Finish | ILikeSheren  ILIKESHEREN  Message contains SHEREN  293  ISHEREN |
| **Input** | **Output** |
| HappyNameDay  Replace p r  Make Lower  Cut 2 23  Sum -2 2  Finish | HarryNameDay  harrynameday  Invalid indexes!  Invalid indexes! |

## 2. Message Translator

Create a program, that **checks** if **inputs** have a **valid command and message** and **encrypt** it. You will receive **n** count of messages. For each message check if it’s valid.

A message is **valid** when:

* The command is **surrounded by** '**!**', **start** with a **uppercase** letter, **followed** **only** by **lowercase** letters.
* The command Is **mininum 3 characters long**
* There is a **colon** after the command.
* There is message **consisting of alphabetical** **letters** between '**[**' and '**]**'.
* It has to be **minimum 8** characters long.

**Example for a valid message:**

**"!Send!:[IvanisHere]"**

You must **check** if the **message** is **valid** and if it **is** - **encrypt** it, if it **isn’t** - **print** the following **message**:

**"The message is invalid"**

**Encrypting** a **message** means to **take** **all** **letters** from the message and **turn** them **into** **ASCII** **numbers**. After successful encrypt, print it in the following format:

**{command}: {number1} {number2} {number3} (…)**

**Note: Encrypt only the text in the message. If you have "[Ivan is Here]", the part that you need to encrypt is "Ivan is Here".**

### Input

* You receive a line - **input** that you have to **check** if it has a **valid** **message**.

### Output

* Print the result in format described above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  !Send!:[IvanisHere]  \*Time@:[Itis5amAlready] | Send: 73 118 97 110 105 115 72 101 114 101  The message is invalid |
| 3  go:[outside]  !drive!:YourCarToACarWash  !Watch!:[LordofTheRings] | The message is invalid  The message is invalid  Watch: 76 111 114 100 111 102 84 104 101 82 105 110 103 115 |

## 3. Nikulden’s meals

*Now you want to find out which meals your guests liked and how many meals they didn't like*

Create a program that keeps information about guests liked and unliked meals.

You will be receiving **lines** with commands until you receive the **"Stop"** command. The **possible** commands are:

* **"Like-{guest}-{mea1}":**
  + Add the **{mea1}** to the **{guest}’s** collection of meals.
  + If the guest **doesn't** exist, **add** it to your record.
  + If the guest **already has the meal** in his collection, don’t add it.
* **"Unlike-{guest}-{meal}":**
  + **Remove**  the meal of the given guest’s collection and print:

**"{Guest} doesn't like the {meal}."**

**You must keep the count of unliked meals!**

* + If the guest doesn’t exist, print:

**"{Guest} is not at the party."**

* + If the guest doesn’t have the meal at the like list, print:

**"{Guest} doesn't have the {meal} in his/her collection."**

In the end, you have to **print the guests with their liked meals** sorted in **descending order** by each guest **meals** **count** and **then by** their **names** in **ascending** order. Then print the count of **unliked meals** in the format below

**{Guest1}: {meal1}, {meal2}, {meal3}**

**{Guest2}: {meal1}, {meal2}**

**...**

**Unliked meals: {count of all unliked meals}**

### Input

* You will be receiving linesuntil you receive the **"Stop"** command.
* The input will **always** be **valid**.

### Output

* **Print** the guests with their **meals** in the **format** described above.
* **Print** the count of **unliked meals** in the **format** described above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Like-Krisi-shrimps  Like-Krisi-soup  Like-Misho-salad  Like-Pena-dessert  Stop | Krisi: shrimps, soup  Misho: salad  Pena: dessert  Unliked meals: 0 |
| **Input** | **Output** |
| Like-Krisi-shrimps  Unlike-Vili-carp  Unlike-Krisi-salad  Unlike-Krisi-shrimps  Stop | Vili is not at the party.  Krisi doesn't have the salad in his/her collection.  Krisi doesn't like the shrimps.  Krisi:  Unliked meals: 1 |

# [Final Exam - 07 December 2019 Group 1](https://judge.softuni.bg/Contests/1928/Programming-Fundamentals-Final-Exam-07-December-2019-Group-1)

## 1. Email Validator

*Your friend has hired you to help him with his website creation. Your current task is to create an Email Validator.*

Create a program that manipulates a string and makes it suitable for an Email. First, you are going to **receive** **the email** that the user wants to use, then you will receive commands.

You will be receiving **commands** until the "**Complete**" command. There are **six** possible commands:

* **"Make Upper"**
  + **Replace** all letters with upper case, then **print** the result.
* **"Make Lower"**
  + **Replace** all letters with lower case, then **print** the result.
* "**GetDomain {count}**
  + Print **the last {count}** characters of the Email**.**
* "**GetUsername"**
  + Print **the substring** from the **start of the Email** until the **@ symbol**.
  + If the Email doesn’t contain the **@** symbol, **print:**

**"The email {email} doesn't contain the @ symbol."**

* "**Replace {char}"**
  + **Replace** all occurences of the **{char}** with **a dash "-"** and **print** the result.
* "**Encrypt"**
  + **Get** the **ASCII** value of each symbol. Print the result on a single line separated by a single space

### Input

* On the **1st line** you are going to receive the **Email in the form of a string**.
* On the next **lines**, until the **"Complete"** command is received, you will be receiving commands.

### Output

* **Print** the **output** of every **command** in the **format** **described** **above**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Mike123@somemail.com  Make Upper  GetDomain 3  GetUsername  Encrypt  Complete | MIKE123@SOMEMAIL.COM  COM  MIKE123  77 73 75 69 49 50 51 64 83 79 77 69 77 65 73 76 46 67 79 77 |
| **Input** | **Output** |
| AnotherMail.com  Make Lower  GetUsername  Replace a  Complete | anothermail.com  The email anothermail.com doesn't contain the @ symbol.  -notherm-il.com |

## 2.Registration

Create a program, that **checks** if **registrations** are **valid**. A registration consists of a **Username** and a **Password.** On the **first** line you will **receive** a **number** that **indicates** how **many** **inputs** you will **receive** on the **next** lines**.**

A registration is **valid** when:

* The username is surrounded by **"U$"**
* The username needs to be **minimum 3** characters long, **start** with an **uppercase** letter, **followed** **only** by **lowercase** letters
* The password is surrounded by "**P@$"**
* The password needs to start with **minimum 5** alphabetical letters **(not including digits)** and must end with **a digit**

**Example for a valid registration:**

**"U$MichaelU$P@$asdqwe123P@$"**

You must **check** if the **registration** is **valid** and if it **is** print:

**"Registration was successful"**

**"Username: {Username}, Password: {Password}"**

If it **isn’t** - **print** the following **message**:

**"Invalid username or password"**

In the end print the count of successful registrations:

**"Successful registrations: {successfulRegistrationsCount}"**

### Input

* On the **first** line - **n** - the count of inputs.
* On the **next** **n** lines - **input** that you have to **check** if it has a **valid** **registration**.

### Output

* Print all results from each input, each on a new line.
* In the end print the count of successful registrations

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 3  U$MichaelU$P@$asdqwe123P@$  U$NameU$P@$PasswordP@$  U$UserU$P@$ad2P@$ | Registration was successful  Username: Michael, Password: asdqwe123  Invalid username or password  Invalid username or password  Successful registrations: 1 | We have 3 input lines to check. The first one follows the rules and is valid. The second one doesn’t because the password doesn’t end with a digit. The third one is not valid because the password is too short. |
| 2  U$TommyU$P@$asdqwe123P@$  Sara 1232412 | Registration was successful  Username: Tommy, Password: asdqwe123  Invalid username or password  Successful registrations: 1 |  |

## 3. Inbox Manager

Create a program that manages **users** and **Emails** **sent** by **users**. You need to keep information about their **username** and their **sent** Emails. **The Emails are represented as strings.** You will be receiving **lines** with commands separated by "**->**" until you receive the **"****Statistics"** command. There are three **possible** commands:

* **"Add->{username}":**
  + **Check** if the username **exists** and if **it does print:**

**"{username} is already registered"**

If it doesn’t exist, then add the user to the collection of users.

* **"Send->{username}->{Email}"**
  + **Add** the **{Email} to the {username}'s collection of sent Emails.**
* **"Delete->{username}":**
  + Delete **the given user**, **if** he **exists**. If the user **doesn’t exist, print:**

"**{username} not found!**"

In the end, you have to **print the count of users, each user** with his/her **Emails.** Users need to be sorted in **descending order** by the **count of sent Emails** and **then by** their **username** in **ascending** order in the following format:

**Users count: {count}**

**{username}**

**- {Email1}**

**- {Email2}**

**- {Emailn}**

### Input

* You will be receiving linesuntil you receive the **"Statistics"** command.
* The commands will be in the format described above.

### Output

* Print the collection in the format described above after the **"Statistics"** command.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add->Mike  Add->George  Send->George->Hello World  Send->George->Some random test mail  Send->Mike->Hello, do you want to meet up tomorrow?  Send->George->It would be a pleasure  Send->Mike->Another random test mail  Statistics | Users count: 2  George  - Hello World  - Some random test mail  - It would be a pleasure  Mike  - Hello, do you want to meet up tomorrow?  - Another random test mail |
| **Comments** | |
| First we receive our users. Since they are not already in our collection, we add them. Then they start sending emails and in the end we print the output in the described format. | |
|  | |
| Add->Mike  Add->George  Send->George->Hello World  Send->George->Your loan is overdue  Add->Mike  Send->Mike->Hello, do you want to meet up tomorrow?  Delete->Peter  Send->George->I'm busy  Send->Mike->Another random test mail  Delete->George  Statistics | Mike is already registered  Peter not found!  Users count: 1  Mike  - Hello, do you want to meet up tomorrow?  - Another random test mail |
|  | |

# [Final Exam Retake - 9 August 2019](https://judge.softuni.bg/Contests/1767/Programming-Fundamentals-Final-Exam-Retake-9-August-2019)

## 1. Username

*Pesho has decided to finally make an account on social media. His problem is that too many people are called Pesho too, so he needs a program that will help him generate an original username and he is asking you for help.*

First, you are going to **receive** **the username** that he wants to use in the first place, then commands.

You will be receiving **commands** until the "**Sign up**" command. There are **six** possible commands:

* **"Case {lower/upper}"**
  + **Replace** all letters with lower case or with upper case, then **print** the result.
* "**Reverse {startIndex} {endIndex}"**
  + **Reverse** the substring from the **startIndex** until the **endIndex**, then **print** it. Do **NOT** change it in the username.

**Note:** Check if the indexes are valid. If they aren’t - skip the command. The indexes are inclusive.

* "**Cut {substring}"**
  + **Check** if the string **contains**  the **substring** and if yes, **cut** it out and **print** the result.
  + If the string doesn’t contain the given substring, **print:**

**"The word {string} doesn't contain {substring}."**

* "**Replace {char}"**
  + **Replace** all occurences of **char** with **astericks** (**\***) and **print** the result.
* "**Check {char}"**
  + In order for a username to be **valid,** it must contain the given **char.**
  + If the password is **valid**, print **"Valid"**. If it is **not valid**, print: **"Your username must contain {char}."**

### Input

* On the **1st line** you are going to receive the **string**.
* On the next **lines**, until the **"Sign up"** command is received, you will be receiving commands.
* All commands are case **sensitive**.

### Output

* **Print** the **output** of every **command** in the **format** **described** **above**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Pesho  Case lower  Cut ES  Check @  Sign up | pesho  The word pesho doesn't contain ES.  Your username must contain @. |
| **Input** | **Output** |
| ThisIsMyString  Reverse 1 4  Replace i  Cut My  Sign up | Isih  Th\*sIsMyStr\*ng  Th\*sIsStr\*ng |

## 2. Password

*Now that Pesho has a username, he needs to think of a secure password. His trouble is that the social media he has chosen has special requirements for valid passwords. So that the users stay secure when making an account, the system uses an ecryption to check and store passwords.*

Create a program, that **checks** if **inputs** are a **valid password** and **encrypt** it. On the **first** line you will **receive** a **number** that **indicates** how **many** **inputs** you will **receive** on the **next** lines**.**

A password is **valid** when:

* It **starts** with a **group** of  **symbols** and **ends** with the **same symbols (the same length)**
* There is a **greater than sign (>)** after the first group and a **less than sign (<)** before the last one
* In between the greater than sign and the less than sign there are **four** **groups** (each of **three** characters), separated by pipe ("**|**")
  + The first group consists only of **numbers**
  + The second group – only **lower case letters**
  + The third one – only **upper case letters**
  + The fourth one – all **symbols except '<' and '>'**

**Example for a valid message:**

**"$$$>312|dfe|KFE|@!#<$$$"**

You must **check** if the **password** is **valid** and if it **is** - **encrypt** it, if it **isn’t** - **print** the following **message**:

**"Try another password!"**

**Encrypting** a **password** means to **take** **all** **numbers, letters and symbols from the middle four groups** and **concatenatе** them. After successful encrypt, print it in the following format:

**Password: {encrypted password}**

### Input

* On the **first** line - **n** - the count of inputs.
* On the **next** **n** lines - **input** that you have to **check** if it has a **valid** **password**.

### Output

* Print all results from each input, each on a new line.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 3  ##>00|no|NO|!!!?<###  ##>123|yes|YES|!!!<##  $$<111|noo|NOPE|<<>$$ | Try another password!  Password: 123yesYES!!!  Try another password! | The first one doesn’t start and end with the same amount of '#' and the count of characters in each group is different than 3. The second one is correct. The third one uses the wrong '<' and '>' and the group containing "<<" can contain everything except '<' and '>'. |
| 5  aa>111|mqu|BAU|mqu<aa  ()>111!aaa!AAA!^&\*<()  o>088|abc|AAA|\*\*\*<o  asd>asd|asd|ASD|asd<asd  \*>088|zzzz|ZzZ|123<\* | Password: 111mquBAUmqu  Try another password!  Password: 088abcAAA\*\*\*  Try another password!  Try another password! |  |

### 3. Followers

*Now that Pesho has successfully created an account, he wants to connect with other users and gain as many followers, likes and comments as possible.*

Create a program that keeps information about Pesho's **followers**, **likes** and **comments.** Keep a record of the followers, each with the likes and comments Pesho has received from them.

You will be receiving **lines** with commands until you receive the **"Log out"** command. There are four **possible** commands:

* **"New follower: {username}":**
  + **Add** the **username**, to your **records** (with 0 likes and 0 comments). **If** person with the given **username** already **exists ignore** **the line**.
* **"Like: {username}: {count}":**
  + If the username **doesn't** exist, **add** it to your records with the given count of likes.
  + If the username **exist**, **increase** the count of likes with the given count.
* **"Comment: {username}":**
  + If the username **doesn't** exist, **add** it to your records with **1 comment**.
  + If the username **exists**, **increase** the count of commens with **1**.
* **"Blocked: {username}":**
  + **Delete** all records of the given username. If it doesn’t exist, print:

**"{Username} doesn't exist."**

In the end, you have to **print the count of followers, each follower** with his/her **likes and comments** (the **sum** of **likes** and **comments**) sorted in **descending order** by the **likes** and **then by** their **username** in **ascending** order in the following format:

**{count} followers**

**{username}: {likes+comments}**

**{username}: {likes+comments}**

**...**

#### Input

* You will be receiving linesuntil you receive the **"Log out"** command.
* The input will **always** be **valid**.

#### Output

* Print the users with their **likes** in the **format** described above.

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| New follower: gosho  Like: gosho: 5  Comment: gosho  New follower: gosho  New follower: tosho  Comment: gosho  Comment: tosho  Comment: pesho  Log out | 3 followers  gosho: 7  pesho: 1  tosho: 1 |
| **Input** | **Output** |
| Like: A: 3  Comment: A  New follower: B  Blocked: A  Comment: B  Like: C: 5  Like: D: 5  Log out | 3 followers  C: 5  D: 5  B: 1 |

# [Final Exam - 03 August 2019 Group 2](https://judge.softuni.bg/Contests/1749/Programming-Fundamentals-Final-Exam-03-August-2019-Group-2)

## 1. String Manipulator - Group 2

Create a program that executes changes over a string. First, you are going to **receive** **the string**, then commands.

You will be receiving **commands** until the "**Done**" command. There are **six** possible commands:

* **"Change {char} {replacement}"**
  + **Replace** all occurences of **{char}** with **{replacement}**, then **print** the **string**.
* "**Includes {string}"**
  + **Check** if the string **includes** with **{string}** and **print** "**True**/**False**".
* "**End {string}"**
  + **Check** if the string **ends** with **{string}** and **print** "**True**/**False**".
* "**Uppercase"**
  + Make the **whole** **string** **uppercased**, then **print** it.
* "**FindIndex {char}"**
  + Find the **first index of {char}**, then **print** it.
* "**Cut {startIndex} {length}"**
  + **Remove** all **characters** from the **string** **except** for those **starting** from **{startIndex}** and the **next** **{length}** characters, then **print** it.

### Input

* On the **1st line** you are going to receive the **string**.
* On the next **lines**, until the **"Done"** command is received, you will be receiving commands.
* All commands are case **sensitive**.
* The **input** will **always** be **valid**.

### Output

* **Print** the **output** of every **command** in the **format** **described** **above**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| //Th1s 1s my str1ng!// Change 1 i Includes string End my Uppercase FindIndex I Cut 5 5 Done | //This is my string!//  True  False  //THIS IS MY STRING!//  4  S IS |

## 2. Message Encrypter

Create a program, that **checks** if **inputs** have a **valid message** and **encrypt** it. On the **first** line you will **receive** a **number** that **indicates** how **many** **inputs** you will **receive** on the **next** lines**.**

A message is **valid** when:

* It is in the **end** of the input
* It **starts** with a **tag**, which is **surrounded** by either '**\***' or '**@**' (but **not both** at the same time), the tag itself has to be **minimum 3** characters long, **start** with a **uppercase** letter, **followed** **only** by **lowercase** letters
* There is a **colon** and a single **white space** after the tag
* There are **3 groups** consisting of **letters** between '**[**' and '**]**', followed by a **pipe** ('**|**')

**Example for a valid message:**

**"\*Request\*: [I]|[s]|[i]| "**

You must **check** if the **message** is **valid** and if it **is**- **decrypt** it, if it **isn’t**- **print** the following **message**:

**"Valid message not found!"**

**Encrypting** a **message** means to **take** **all** **letters** and **turn** them **into** **ASCII** **numbers**. After successful encrypt, print it in the following format:

**{tag}: {number1} {number2} {number3} (…)**

### Input

* On the **first** line - **n** - the count of inputs.
* On the **next** **n** lines - **input** that you have to **check** if it has a **valid** **message**.

### Output

* Print all results from each input, each on a new line.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 3  \*Request\*: [I]|[s]|[i]|  \*Taggy@: [73]|[73]|[73]|  Should be valid @Taggy@: [v]|[a]|[l]| | Request: 73 115 105  Valid message not found!  Taggy: 118 97 108 | We have 3 input lines to check. The first one follows the rules and is valid. The second one doesn’t because the tag is surrounded by both '\*' and '@'. The third one has a valid message and is in the end of the input. |
| 3  @Taggy@: [i]|[n]|[v]|[a]|[l]|[i]|[d]| this shouldn’t be valid  \*tAGged\*: [i][i][i]|  Should be invalid @Taggy@: [v]|[a]|[l]|[l]|[l]| | Valid message not found!  Valid message not found!  Valid message not found! |  |

## 3. Battle Manager

Create a program that manages battles. You need to keep information about **people**, the **health** and **energy** they have. You will be receiving **lines** with commands until you receive the **"Results"** command. There are three **possible** commands:

* **"Add:{personName}:{health}:{energy}":**
  + **Add** the **person**, his/her **health** and **energy** to your **records**. **If** person with the given name already **exists**, just increase the **health** of the **person** with the **current** one that is **given**.
* **"Attack:{attackerName}:{defenderName}:{damage}":**
  + **Check** if both people **exist** and if **they do**, **reduce** the **defender’s** **health** with the **damage** **given**. If the defender’s health reaches **0** or **less**, the **person** is **disqualified,** and you need to **remove** him/her from your **records** and **print** the following **message**:
    - **"{defenderName} was disqualified!"**
  + You also have to **reduce** the **attacker’s** **energy** **by** **1**. If it reaches **0**, he/she is **disqualified,** and you need to **remove** him/her from your **records** and **print** the following **message**:
    - **"****{attackerName} was disqualified!"**
* **"Delete:{username}":**
  + Delete **all** records of the **given user**, **if** he **exists**. If "**All**" is **given as username** - delete **all records** you have.

In the end, you have to **print the count of people left, each person** with his/her **health** and **energy** sorted in **descending order** by the **health** and **then by** their **name** in **ascending** order in the following format:

**People count: {count}**

**{personName} - {health} - {energy}**

**{personName} - {health} - {energy}**

### Input

* You will be receiving linesuntil you receive the **"Results"** command.
* The **health** is an **integer** number in the range [1...100000].
* The **energy** is an **integer** number in the range [1...100].
* The input will **always** be **valid**.

### Output

* Print the appropriate message after the **"Attack"** command, **if** someone is **disqualified**.
* Print the people with their **health and energy** in the **format** described above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add:Mark:1000:5 Add:Clark:1000:3 Attack:Clark:Mark:500 Add:Allison:2500:5 Attack:Clark:Mark:300 Add:Charlie:4000:10 Attack:Clark:Mark:500 Results | Mark was disqualified!  Clark was disqualified!  People count: 2  Charlie - 4000 - 10  Allison - 2500 - 5 |
| **Comments** | |
| First, we receive the "**Add:Mark:1000:5**" command, so we **add** the person **Mark** in our **records** with his **health** (1000) and **energy** (5). Then we do the **same** for **Clark**. Afterwards **Clark attacks Mark multiple times**, and **Mark’s health** is **below 0** and **Clark’s energy** drops to **0**, meaning **both** of them are **disqualified**. Meanwhile, **Allison** and **Charlie** were **added**. In the end, we **print** the results- **2** people, then the collection **ordered** as described above. | |
|  | |
| Add:Bonnie:3000:5  Add:Kent:10000:10 Add:Johny:4000:10 Attack:Johny:Bonnie:400 Add:Chicken:1000:1 Add:Rabbit:3000:5 Add:Buggy:1259:10  Delete:Kent Attack:Chicken:Rabbit:1000 Results | Chicken was disqualified!  People count: 4  Johny - 4000 - 9  Bonnie - 2600 - 5  Rabbit - 2000 - 5  Buggy - 1259 - 10 |
|  | |
| Add:Bonnie:3000:5 Add:Johny:4000:10 Delete:All  Add:Bonnie:3333:3 Results | People count: 1  Bonnie - 3333 - 3 |

# [Final Exam - 03 August 2019 Group 1](https://judge.softuni.bg/Contests/1748/Programming-Fundamentals-Final-Exam-03-August-2019-Group-1)

## 1. String Manipulator - Group 1

Create a program that executes changes over a string. First, you are going to **receive** **the string**, then commands.

You will be receiving **commands** until the "**End**" command. There are **six** possible commands:

* **"Translate {char} {replacement}"**
  + **Replace** all occurences of **{char}** with **{replacement}**, then **print** the **string**.
* "**Includes {string}"**
  + **Check** if the string **includes** **{string}** and **print** "**True**/**False**".
* "**Start {string}"**
  + **Check** if the string **starts** with **{string}** and **print** "**True**/**False**".
* "**Lowercase"**
  + Make the **whole** **string** **lowercased**, then **print** it.
* "**FindIndex {char}"**
  + Find the **last index of {char}**, then **print** it.
* "**Remove {startIndex} {count}"**
  + Remove **{count}** characters from the string, starting from **{startIndex}**, then **print** it.

### Input

* On the **1st line** you are going to receive the **string**.
* On the next **lines**, until the **"End"** command is received, you will be receiving commands.
* All commands are case **sensitive**.
* The **input** will **always** be **valid**.

### Output

* **Print** the **output** of every **command** in the **format** **described** **above**.

|  |  |
| --- | --- |
| **Input** | **Output** |
| //Thi5 I5 MY 5trING!// Translate 5 s Includes string Start //This Lowercase FindIndex i Remove 0 10 End | //This Is MY strING!//  False  True  //this is my string!//  16  my string!// |

## 2. Message Decrypter

Create a program, that **checks** if **inputs** have a **valid message** and **decrypt** it. On the **first** line you will **receive** a **number** that **indicates** how **many** **inputs** you will **receive** on the **next** lines**.**

A message is **valid** when:

* There is **nothing** else **before** and **after it**
* It **starts** with a **tag**, which is **surrounded** by either '**$**' or '**%**' (but **not both** at the same time), the tag itself has to be **minimum 3** characters long, **start** with a **uppercase** letter, **followed** **only** by **lowercase** letters
* There is a **colon** and a single **white space** after the tag
* There are **3 groups** consisting of **numbers** between '**[**' and '**]**', followed by a **pipe** ('**|**')

**Example for a valid message:**

**"$Request$: [73]|[115]|[32]|"**

You must **check** if the **message** is **valid** and if it **is**- **decrypt** it, if it **isn’t** - **print** the following **message**:

**"Valid message not found!"**

**Decrypting** a **message** means to **take** **all** **numbers** and **turn** them **into** **ASCII** **symbols**. After successful decrypt, print it in the following format:

**{tag}: {decryptedMessage}**

### Input

* On the **first** line - **n** - the count of inputs.
* On the **next** **n** lines - **input** that you have to **check** if it has a **valid** **message**.

### Output

* Print all results from each input, each on a new line.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 4  $Request$: [73]|[115]|[105]|  %Taggy$: [73]|[73]|[73]|  %Taggy%: [118]|[97]|[108]|  $Request$: [73]|[115]|[105]|[32]|[75]| | Request: Isi  Valid message not found!  Taggy: val  Valid message not found! | We have 3 input lines to check. The first one follows the rules and is valid. The second one doesn’t because the tag is surrounded by both '%' and '$'. The third one has a valid message and is in the beginning of the input. The last one is invalid because it has more than 3 groups of numbers. |
| 3  This shouldnt be valid%Taggy%: [118]|[97]|[108]|  $tAGged$: [97][97][97]|  $Request$: [73]|[115]|[105]|true | Valid message not found!  Valid message not found!  Valid message not found! |  |

## 3. Messages Manager

Create a program that manages **messages** **sent** and **received** of **users**. You need to keep information about **username**, their **sent** and **received** messages. You will **receive** the **capacity** of **possible** **messages** **kept** **at once per user**. You will be receiving **lines** with commands until you receive the **"Statistics"** command. There are three **possible** commands:

* **"Add={username}={sent}={received}":**
  + **Add** the **username**, his/her **sent** and **received** messages to your **records**. **If** person with the given **username** already **exists ignore** **the line**.
* **"Message={sender}={receiver}":**
  + **Check** if both usernames **exist** and if **they do**, **increase** the **sender’s** **sent messages** by 1 and the **receiver’s received messages** by 1. If anyone **reaches** the **capacity** (**first check the sender**), he/she should be **removed** fromthe **record** and you should **print** the following message:
    - **"{username} reached the capacity!"**
* **"Empty={username}":**
  + Delete **all** records of the **given user**, **if** he **exists**. If "**All**" is **given as username** - delete **all records** you have.

In the end, you have to **print the count of users, each person** with his/her **messages** (the **count** of both **sent** and **received**) sorted in **descending order** by the **received messages** and **then by** their **username** in **ascending** order in the following format:

**Users count: {count}**

**{username} - {messages}**

**{username} - {messages}**

### Input

* On the **first** **line**, you will **receive** the **capacity** - an **integer** number in the range [1-10000].
* You will be receiving linesuntil you receive the **"Statistics"** command.
* The **initial messages** (**sent** and **received**)will **always** be **below** the **capacity**.
* The input will **always** be **valid**.

### Output

* Print the appropriate message after the **"Message"** command, **if** someone reaches the capacity.
* Print the users with their **messages** in the **format** described above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 10 Add=Mark=5=4 Add=Clark=3=5 Add=Berg=9=0 Add=Kevin=0=0 Message=Berg=Kevin Statistics | Berg reached the capacity!  Users count: 3  Clark - 8  Mark - 9  Kevin - 1 |
| **Comments** | |
| First, we **receive** the **capacity** (10). Then we **start** **receiving** **commands**. The **first four commands** are for **adding new users**, so we do it. Then we have the command “**Message=Berg=Kevin**” and **Berg** **reached** the **capacity**, so we **remove** him, but **Kevin** has only his **received** messages **incremented**. When we receive the “**Statistics**” command, we **print** the **output** as described **above**. | |
|  | |
| 20 Add=Mark=3=9  Add=Berry=5=5 Add=Clark=4=0 Empty=Berry Add=Blake=9=3 Add=Michael=3=9 Add=Amy=9=9 Message=Blake=Amy Message=Michael=Amy Statistics | Amy reached the capacity!  Users count: 4  Mark - 12  Michael - 13  Blake - 13  Clark - 4 |
|  | |
| 12  Add=Bonnie=3=5 Add=Johny=4=4  Empty=All  Add=Bonnie=3=3 Statistics | Users count: 1  Bonnie - 6 |

# [Final Exam Preparation - 24 July 2019](https://judge.softuni.bg/Contests/1759/Programming-Fundamentals-Final-Exam-Preparation-24-July-2019)

### 1. Concert

*Mandy was hired to accept the group applications for the upcoming concert. She thought that it would be easy but now she is in trouble. She needs a programmer to make an application that will help her to safe the concert. She needs you.*

You will read commands until you receive **"start of concert"** command.

There are **two types** of commands:

* "**Add;** **{bandName}; {member 1}, {member 2}, {member 3}"** - applies a band and a **list of members** to the concert. All members must be **unique** so don't add duplicates. If you receive the **same band** twice add only those members that **aren't present** in the list.
* **"Play; {bandName}; {time}"** – the band with the given name plays an **amount of time** on the stage. If you receive a **band** that has **already** **applied** in the concert, just **increase** the band **time.**

If in both commands the band **does not exist**, add it.

At the end you have to print the **total time** and the bands ordered by the **time** on stage in **descending** order, then by **band name** in **ascending** order.

Also the **final input line** will be "**{bandName}**" and you have to print **all members** for this band in **insertion order**.

#### Input / Constraints

* The **time** of the bands – **integer** in range **[0 – 231 - 1]**
* There will always be at least one **member** in the group
* The bands will always have **time** on stage
* The final input line will always contain an **existing** band name
* Input will always be valid and in the range specified. You don’t need to check it

#### Output

**Total time: {totalTime}**

**{firstBandName} -> {firstBandTime}**

**{secondBandName} -> {secondBandTime}**

**{thirdBandName} -> {thirdBandTime}**

**…**

**{bandName}**

**=> {firstMemberName}**

**=> {secondMemberName}**

**=> {thirdMemberName}**

**…**

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Play; The Beatles; 2584  Add; The Beatles; John Lennon, Paul McCartney, George Harrison, Ringo Starr  Add; Eagles; Glenn Frey, Don Henley, Bernie Leadon, Randy Meisner  Play; Eagles; 1869  Add; The Rolling Stones; Brian Jones, Mick Jagger, Keith Richards  Add; The Rolling Stones; Brian Jones, Mick Jagger, Keith Richards, Bill Wyman, Charlie Watts, Ian Stewart  Play; The Rolling Stones; 4239  start of concert  The Rolling Stones | Total time: 8692  The Rolling Stones -> 4239  The Beatles -> 2584  Eagles -> 1869  The Rolling Stones  => Brian Jones  => Mick Jagger  => Keith Richards  => Bill Wyman  => Charlie Watts  => Ian Stewart |
| Add; The Beatles; John Lennon, Paul McCartney  Add; The Beatles; Paul McCartney, George Harrison  Add; The Beatles; George Harrison, Ringo Starr  Play; The Beatles; 3698  Play; The Beatles; 3828  start of concert  The Beatles | Total time: 7526  The Beatles -> 7526  The Beatles  => John Lennon  => Paul McCartney  => George Harrison  => Ringo Starr |

### 2. Song Encryption

*Now that you've helped Mandy to accept the group applications it's time to assist her with a security problem. You are tasked to encrypt all songs from the set list so that if someone steals it they won't be able to leak it online.*

Your task is to write a program that encrypts information about artists and their songs.

**Until** you receive the command **"end"** you should read lines in following format :"**{artist}:{song}",** and check if it's **valid**, considering the following rules:

* **Artist** – **starts** with **capital letter**, followed by **lowercase letters.**
  + It can also **contains apostrophe ( ' ),** and **whitespace " ";**

**Valid group name:** Red hot chili peppers, Eminem, Guns n' roses

**Invalid group name**: ReD Hot CiLly PePers, sLipKnot, guns n'roses

* **Song – contains only capital letters, and whitespaces.**

**Valid songs:** BACK IN BLACK, BLEED IT OUT, KILLSHOT

**Invalid songs:** #BaCk IN black, BLEED $IT$ OUTt, &KILLSHoT

After you validate the lines of an input, you should encrypt the information. In order to do that, you have to follow the rules below:

* **First you need to find a key for encryption.**
  + Your key is the **length** of the **artist (e.g. "Eminem" – key: 6)**
* You have to **increment** the **ASCII value** of each symbol of the input, with the **key length**

**(**e.g. "char" **'a'** -> **'g')**

* + Be careful if your key length is **bigger** than the ASCII value of a letter **'z'** or **'Z'**. In this case you should start from a letter **'a'. (**e.g. key:6 letter – '**x**', encrypted letter – '**d**'**)**
* You should **NOT ENCRYPT** the following characters: **whitespaces**, and **apostrophes**
* You also should **replace** **':'** with the sign **'@'**

#### Input / Constraints

Until you receive **"end"** command you should read from the console.

* **Line of input** – Artist name and song, **separated** by **":"**, containing only **ASCII symbols.**
* Allowed working time for your program: 0.1 seconds.
* Allowed memory: 16 MB.

#### Output

After every line of input, you should print:

* **If** line is valid – **encrypted information** in following format:

**"Successful encryption:** **{encryptedArtist}@{encryptedSong}".**

* **If** line is not valid – print the following message: **"Invalid input!"**

#### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Eminem:VENOM  Linkin park:NUMB  Drake:NONSTOP  Adele:HELLO  end | Successful encryption: Ksotks@BKTUS  Successful encryption: Wtyvty alcv@YFXM  Successful encryption: Iwfpj@STSXYTU  Successful encryption: Fijqj@MJQQT | All lines of input **are valid**, so we  encrypt the information, change the charracter ":" with the sign **"@"**, and print the output of encription.  *Example*: **Eminem-> key 6**, adding a key to the ASCII value of each charracter except the **whitespace**, **apostrophe**, and our **delimiter**(":") and receive an encrypted name – **Ksotks@BKTUS**, then we do the same with the song. |
| **Input** | **Output** | **Comments** |
| Michael Jackson:ANOTHER PART OF ME  Adele:ONE AND ONLY  Guns n'roses:NOVEMBER RAIN  Christina Aguilera: HuRt  end | Invalid input!  Successful encryption: Fijqj@TSJ FSI TSQD  Successful encryption: Sgze z'daeqe@ZAHQYNQD DMUZ  Invalid input! | First line in not valid, because in the name of Michael Jackson we have more than one capital letter. Next two are valid, and the last is not valid, because the song does not contain only capital letters. |

### 3. The Isle of Man TT Race

*This year’s* [*Isle of Man TT Race*](https://en.wikipedia.org/wiki/Isle_of_Man_TT) *is going to be around Douglas and your job is to find the exact coordinates for it and the names of the racers. Every racer starts from a different place. You’re going to receive the coordinates in the form of a* [*geohash*](https://en.wikipedia.org/wiki/Geohash) *code.*

**

Write a program that decrypts messages. You’re going to receive a few notes that contain the following information:

* **Name of racer**
  + Consists only of letters. It is surrounded from the both sides by any of the following symbols – **"#, $, %, \*, &". Both symbols** – in the **beginning** and at the **end** of the name should **match**.
* **Length of geohashcode**
  + Begins after the **"="** sign and it is consisted only of numbers.
* **Encrypted geohash code**
  + Begins after these symbols - **“!!”**. It may contain anything and the message always ends with it.

**Examples for valid input:**

#SteveHislop#=16!!tv5dekdz8x11ddkc  
**Examples of invalid input:**

%GiacomoAgostini$=7!!tv58ycb – The length is the same, but the name is not surrounded by **matching** **signs**.

$GeoffDuke$=6!!tuvz26n35dhe4w4 – The length doesn't **match** the **lengh** of the code.

&JoeyDunlop&!!tvndjef67t=14 – The length should be **before** the code.

The information must be in the **given order**, otherwise it is considered **invalid**. The **geohash code** you are looking for is with length **exactly** **as much as the given length in the message**. To **decrypt** the code you need to **increase** the value of **each symbol** from the geohashcode with the **given length**. If you find a **match**, you have to **print** the following message:

"**Coordinates found! {nameOfRacer} -> {geohashcode}**"

and stop the program. Otherwise, after every **invalid** message print:

"**Nothing found!**"

#### Input / Constraints

* You will be receiving strings.
* There will always be a valid message.

#### Output

* If you find the right coordinates, print: "Coordinates found! {nameOfRacer} -> {geohashcode}".
* Otherwise, print: "Nothing found!".

#### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| %GiacomoAgostini%=7!!hbqw  &GeoffDuke\*=6!!vjh]zi  JoeyDunlop=10!!lkd,rwazdr  Mike??Hailwood=5!![pliu  #SteveHislop#=16!!df%TU[Tj(h!!TT[S | Nothing found!  Nothing found!  Nothing found!  Nothing found!  Coordinates found! SteveHislop -> tv5dekdz8x11ddkc |
| **Comments** | |
| The first line matches, but the **length** of the code **doesn't match** the given number, so we print "Nothing found!"  The second line begins with **"&"**, but ends with **"\*"**, so we print "Nothing found!"  The third line is not valid because the **name is not surrounded** with one of the **allowed** **symbols**.  The forth line is not a match, because the name doesn't contain **only** letters.  The fifth line is a match and the length is equal to the given number - 16, so we increase each of the symbols from the code with 16 and print the message in the appropriate format. | |
|  | |
| Ian6Hutchinson=7!!\(58ycb4  #MikeHailwood#!!'gfzxgu6768=11  slop%16!!plkdek/.8x11ddkc  $Steve$=9Hhffjh  \*DavMolyneux\*=15!!efgk#'\_$&UYV%h%  RichardQ^uayle=16!!fr5de5kd | Nothing found!  Nothing found!  Nothing found!  Coordinates found! DaveMolyneux -> tuvz26n35dhe4w4 |



# Final Exam - 14 April 2019 Group II

## 1. The Isle of Man TT Race

*This year’s* [*Isle of Man TT Race*](https://en.wikipedia.org/wiki/Isle_of_Man_TT) *is going to be around Douglas and your job is to find the exact coordinates for it and the names of the racers. Every racer starts from a different place. You’re going to receive the coordinates in the form of a* [*geohash*](https://en.wikipedia.org/wiki/Geohash) *code.*

**

Write a program that decrypts messages. You’re going to receive a few notes that contain the following information:

* **Name of racer**
  + Consists only of letters. It is surrounded from the both sides by any of the following symbols – **"#, $, %, \*, &". Both symbols** – in the **beginning** and at the **end** of the name should **match**.
* **Length of geohashcode**
  + Begins after the **"="** sign and it is consisted only of numbers.
* **Encrypted geohash code**
  + Begins after these symbols - **“!!”**. It may contain anything and the message always ends with it.

**Examples for valid input:**

#SteveHislop#=16!!tv5dekdz8x11ddkc  
**Examples of invalid input:**

%GiacomoAgostini$=7!!tv58ycb – The length is the same, but the name is not surrounded by **matching** **signs**.

$GeoffDuke$=6!!tuvz26n35dhe4w4 – The length doesn't **match** the **lengh** of the code.

&JoeyDunlop&!!tvndjef67t=14 – The length should be **before** the code.

The information must be in the **given order**, otherwise it is considered **invalid**. The **geohash code** you are looking for is with length **exactly** **as much as the given length in the message**. To **decrypt** the code you need to **increase** the value of **each symbol** from the geohashcode with the **given length**. If you find a **match**, you have to **print** the following message:

"**Coordinates found! {nameOfRacer} -> {geohashcode}**"

and stop the program. Otherwise, after every **invalid** message print:

"**Nothing found!**"

### Input / Constraints

* You will be receiving strings.
* There will always be a valid message.

### Output

* If you find the right coordinates, print: "Coordinates found! {nameOfRacer} -> {geohashcode}".
* Otherwise, print: "Nothing found!".

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| %GiacomoAgostini%=7!!hbqw  &GeoffDuke\*=6!!vjh]zi  JoeyDunlop=10!!lkd,rwazdr  Mike??Hailwood=5!![pliu  #SteveHislop#=16!!df%TU[Tj(h!!TT[S | Nothing found!  Nothing found!  Nothing found!  Nothing found!  Coordinates found! SteveHislop -> tv5dekdz8x11ddkc |
| **Comments** | |
| The first line matches, but the **length** of the code **doesn't match** the given number, so we print "Nothing found!"  The second line begins with **"&"**, but ends with **"\*"**, so we print "Nothing found!"  The third line is not valid because the **name is not surrounded** with one of the **allowed** **symbols**.  The forth line is not a match, because the name doesn't contain **only** letters.  The fifth line is a match and the length is equal to the given number - 16, so we increase each of the symbols from the code with 16 and print the message in the appropriate format. | |
|  | |
| Ian6Hutchinson=7!!\(58ycb4  #MikeHailwood#!!'gfzxgu6768=11  slop%16!!plkdek/.8x11ddkc  $Steve$=9Hhffjh  \*DavMolyneux\*=15!!efgk#'\_$&UYV%h%  RichardQ^uayle=16!!fr5de5kd | Nothing found!  Nothing found!  Nothing found!  Coordinates found! DaveMolyneux -> tuvz26n35dhe4w4 |



## 2. Practice sessions

*The racers must practice for the race. Your job is to keep the records of the roads and the time for each lap. The track with the best time will be the chosen one for the finals.*

**

Write a program, that keeps information about **roads** and **the racers** who practice on them. When the practice begins, you’re going to start receiving data until you get the "**END**" message. There are three possible commands:

* "Add->{road}->{racer}"
  + Add the **road** if it **doesn't exist** in your collection and add the **racer** to it.
* "Move->{currentRoad}->{racer}->{nextRoad}"
  + Find the **racer** on the **current road** and move him to the **next one,** only if he **exists** in the **current road.** Both roads will always be **valid** and will **already exist**.
* "Close->{road}"
  + Find the **road** and **remove** it from the sessions, **along** **with** the **racers** on it **if it exists**.

In the end, print all of the **roads** with the **racers** who have practiced and **ordered by the count of the racers in descending order**, **then by** the **roads** in **ascending** order. The output must be in the following format:

**Practice sessions:**

**{road}**

**++{racer}**

**++{racer}**

**++{racer}**

………………………..

### Input / Constraints

* You will be receiving lines of information in the format described above, until you receive the **"END"** command.
* The input will always be in the right format.
* Both **roads** from the "**Move**" command will always be **valid** and you don't need to check them explicitly.

### Output

* Print the **roads** withtheir **racers** in the **format described above**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add->Glencrutchery Road->Giacomo Agostini  Add->Braddan->Geoff Duke  Add->Peel road->Mike Hailwood  Add->Glencrutchery Road->Guy Martin  Move->Glencrutchery Road->Giacomo Agostini->Peel road  Close->Braddan  END | Practice sessions:  Peel road  ++Mike Hailwood  ++Giacomo Agostini  Glencrutchery Road  ++Guy Martin |
| **Comments** | |
| We add racers to the roads they are racing on. When we receive the "**Move**" command, we **check** if **Giacomo Agostini** is on **Glencrutchery Road** and if he is, we **remove** him from it and **add** him to the next one - **Peel road**.  When we receive the "**Close**" command, we **remove** Brandon road and **remove** all its records. In the end we print the **roads** **sorted** by the **count** of **racers** on them and **then by** the **names** of the **roads** in **ascending order**. | |
|  | |
| Add->Glen Vine->Steve Hislop  Add->Ramsey road->John McGuinness  Add->Glen Vine->Ian Hutchinson  Add->Ramsey road->Dave Molyneux  Move->Ramsey road->Hugh Earnsson->Glen Vine  Add->A18 Snaefell mountain road->Mike Hailwood  Add->Braddan->Geoff Duke  Move->A18 Snaefell mountain road->Mike Hailwood->Braddan  Move->Braddan->John McGuiaaaaaaaanness->Glen Vine  Close->A18 Snaefell mountain road  END | Practice sessions:  Braddan  ++Geoff Duke  ++Mike Hailwood  Glen Vine  ++Steve Hislop  ++Ian Hutchinson  Ramsey road  ++John McGuinness  ++Dave Molyneux |



## 3. Competitor Entries

Zip file

# Final Exam - 14 April 2019 Group I

## 1. Arriving in Kathmandu

*Your friend is a mountaineer and he needs your help. Your first task is to find him, so you went to Kathmandu and found some notes at his quarters.*



Write a program that **decrypts messages**, which containinformationaboutcoordinates. You are looking for **names of peaks** in the Himalayas and their [geohash](https://en.wikipedia.org/wiki/Geohash) coordinates. Keep reading lines until you receive the "**Last note**" message.

Here is your **cipher**:

* **Name of the peak**
  + It is consisted of **letters (upper and lower), numbers** and some of the following characters between its letters – "**!**" "**@**" "**#**" "**$**" "**?**". Example for valid names: “!@K?#2!#” (K2).
* **The length of the geohashcode**
  + Begins **after** the "**=**" (equals) sign and is consisted only of numbers.
* **The geohash code**
  + Begins after these symbols – "**<<**", may contain anything and the message always ends with it.

**Examples for valid input:**

"!Ma$$ka!lu!@=9<<ghtucjdhs" – all the components are there – **name of the peek**, **length** of the geohashcode and a **geohashcode**.

"!@Eve?#rest!#=7<<vbnfhfg"

**Examples of invalid input:**

"anna@fg<<jhsd@bx!=4" – **their order is wrong**. The name should be first, the length after and the code last.

"#n...s!n-<<tyuhgf4" – **the length is missing** and the **name contains dots.**

**"**Nan$ga!Parbat=8<<gh2tn – **the** **length** of the geohash code doesn't match the given number.

The **geohash code** you are looking for is with **length** **exactly** as much as the **given length** in the message and the information must be in the **exact given order**, otherwise it is considered **invalid**. If you find it, print the following message:

"**Coordinates found! {nameOfMountain} -> {geohashcode}**"

Otherwise print: “**Nothing found!**” after every **invalid** message.

### Input / Constraints

* You will be receiving strings until you get the “**Last note**” message.

### Output

* If you find the right coordinates, print: "**Coordinates found! {nameOfMountain} -> {geohashcode}**".
* If the message is invalid, print: "**Nothing found!**".

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| !@Ma?na?sl!u@=7<<tv58ycb4845  E!ve?rest=.6<<tuvz26  !K@2.,##$=4<<tvnd  !Shiha@pan@gma##9<<tgfgegu67  !###Anna@pur@na##=16<<tv5dekdz8x11ddkc  Last note | Nothing found!  Nothing found!  Nothing found!  Nothing found!  Coordinates found! Annapurna -> tv5dekdz8x11ddkc |
| **Comments** | |
| The first line is invalid, because the length – **7**, **doesn't** **match** the **length** of the **code**.  The second line is invalid, because the **length** should be consisted **only** of **numbers**.  The third line is invalid, because the name contains **symbols** that are **not** allowed – **".", ",".**  The forth line is invalid, because the **"="** sign before the length is **missing**.  The fifth line is valid, so we print the appropriate message. | |
|  | |
| Ka?!#nch@@en@ju##nga@=3<<thfbghvn  =9Cho?@#Oyu<<thvb7ydht  Nan??ga#Par!ba!t?=16<<twm03q2rx5hpmyr6  Dhau??la#gi@ri?!#=3<<bvnfhrtiuy  Last note | Nothing found!  Nothing found!  Coordinates found! NangaParbat -> twm03q2rx5hpmyr6  Nothing found! |

## 2. On the Way to Annapurna

*You’ve hired a Sherpa and he has a list of supplies you both need to go on the way. He has passed you some notes and you have to order them correctly in a diary before you start circling around the town’s stores.*



Create a program, that lists **stores** and the **items** that can be found in them. You are going to be receiving **commands** with the information you need until you get the "**End**" command. There are **three possible commands**:

* "**Add**->{Store}->{Item}"
  + **Add** the **store** and the **item** in your diary. If the store already **exists**, add just the item.
* **"Add**->{Store}->{Item},{Item1}…,{ItemN}"
  + **Add the store and the items to** your notes. **If the store already exists** in the diary – **add just the items** to it.
* "**Remove**->{Store}"
  + **Remove the store** and its items from your diary, **if it exists**.

In the end, print the collection **sorted by the count of the items** in **descending order** and **then by the names of the stores**, again, **in descending order** in the following format:

**Stores list:**

**{Store}**

**<<{Item}>>**

**<<{Item}>>**

**<<{Item}>>**

### Input / Constraints

* You will be receiving information until the “**END**” command is given.
* There will always be **at least one** store in the diary.
* Input will always be **valid**, there is no need to check it explicitly.

### Output

* Print the list of stores in the format given above.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add->PeakSports->Map,Navigation,Compass  Add->Paragon->Sunscreen  Add->Groceries->Dried-fruit,Nuts  Add->Groceries->Nuts  Add->Paragon->Tent  Remove->Paragon  Add->Pharmacy->Pain-killers  END | Stores list:  PeakSports  <<Map>>  <<Navigation>>  <<Compass>>  Groceries  <<Dried-fruit>>  <<Nuts>>  <<Nuts>>  Pharmacy  <<Pain-killers>> |
| **Comments** | |
| First, we receive the "**Add**" command with a couple of items and we have to add the store and the items to. We keep doing that for each line of input and when we receive the "**Remove**" command, we delete the store and its items from our records. In the end we print the stores sorted by the **count** of their **items** and **then by** their **names**. | |
|  | |
| Add->Peak->Waterproof,Umbrella  Add->Groceries->Water,Juice,Food  Add->Peak->Tent  Add->Peak->Sleeping-Bag  Add->Peak->Jacket  Add->Groceries->Lighter  Remove->Groceries  Remove->Store  END | Stores list:  Peak  <<Waterproof>>  <<Umbrella>>  <<Tent>>  <<Sleeping-Bag>>  <<Jacket>> |