# Programming Fundamentals Final Exam 04.04.2020

## Problem 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 |

# Problem 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@#  @###Brea00D@###  @##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 |

# Programming Fundamentals Final Exam 04.04.2020

## Problem 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 the **maximum value** (**200)**, MP is increased to **200**. (the MP can’t go over the maximum value).
* 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 the **maximum value (100)**, HP is increased to **100** (the HP can’t go over the maximum value).
* 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. | | |

# SoftUni Reception

Every day thousands of students pass by the reception at SoftUni with different questions to ask and the employees have to help everyone by providing all the information and to answer all of the questions.

There are **3 employees** working on the reception all day. Each of them can handle **different number** **of students** **per hour**. Your task is to **calculate how much time** it will take to **answer all the questions** of given number of students.

First you will receive 3 lines with integers, representing **count of students that each of the employee can help per hour.** On the next line you will receive **students count as a single integer**.

**Every forth hour all of the employees have a break, so they don’t work for a hour.** This is the only break for the employees, because they don`t need rest, nor have a personal life. Calculate the time needed to answer all the student's questions and print it in the following format: "Time needed: {time}h."

## Input / Constraints

* On first three lines - **each employee efficiency** - integer in range **[1 - 100]**
* On the fourth line - **students count** – integer in range **[0 – 10000]**
* Input will always be valid and in the range specified

## Output

* Print a single line: "Time needed: {time}h."
* Allowed working **time** / **memory**: **100ms** / **16MB**

## Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 5  6  4  20 | Time needed: 2h. | All employees can answer 15 students per hour. After the first hour there are 5 students left to be answered.  All students will be answered in the second hour. |
| 1  2  3  45 | Time needed: 10h. | All employess can answer 6 students per hour. In the first 3 hours they have answered 6 \* 3 = 18 students. Then they have a break for an hour.  After the next 3 hours there are  18 + 6 \* 3 = 36 answered students.  After the break for an hour, there are only 9 students to answer.  So in the 10th hour all of the student's questions would be answered. |
| 3  2  5  40 | Time needed: 5h. |  |

|  |  |  |
| --- | --- | --- |
| **Input JavaScript** | **Output** | **Comment** |
| ['5','6','4','20'] | Time needed: 2h. | All employees can answer 15 students per hour. After the first hour there are 5 students left to be answered.  All students will be answered in the second hour. |
| ['1','2','3','45'] | Time needed: 10h. | All employess can answer 6 students per hour. In the first 3 hours they have answered 6 \* 3 = 18 students. Then they have a break for an hour.  After the next 3 hours there are  18 + 6 \* 3 = 36 answered students.  After the break for an hour, there are only 9 students to answer.  So in the 10th hour all of the student's questions would be answered. |
| ['3','2','5','40'] | Time needed: 5h. |  |

# 02. Array Modifier

You are given **an array with integers**. Write a program to **modify the elements** after **receive the commands** “**swap**”, “**multiply**” or “**decrease**”.

* “swap {index1} {index2}” take **two elements** and **swap their places**.
* “multiply {index1} {index2}” take **element at the 1st index** and **multiply** **it** **with element at 2nd index**. **Save the product at the 1st index.**
* “decrease” **decreases** **all elements** in the array **with 1**.

## Input

On the **first input line** you will be given **the initial array values** separated by a single space.

On the **next lines** you will receive commands **until** you receive the **command “end”**. The **commands are** as follow:

* “swap {index1} {index2}”
* “multiply {index1} {index2}”
* “decrease”

## Output

**The output** should be printed on the console and consist **element** **of the** **modified array** – **separated by “, “**(**comma and single space**).

## Constraints

* Commands will be: “**swap**”, “**multiply**”, “**decrease**” and “**end**”
* **Elements of the array** will be **integer numbers** in the range **[-231...231]**
* **Count of the array elements** will be in the range **[2...100]**
* **Indexes** **will be always in the range of the array**

## Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 23 -2 321 87 42 90 -123  swap 1 3  swap 3 6  swap 1 0  multiply 1 2  multiply 2 1  decrease  end | 86, 7382, 2369942, -124, 41, 89, -3 | 23 -2 321 87 42 90 -123 – initial values  swap 1(-2) and 3(87) ▼  23 87 321 -2 42 90 -123  swap 3(-2) and 6(-123) ▼  23 87 321 -123 42 90 -2  swap 1(87) and 0(23) ▼  87 23 321 -123 42 90 -2  multiply 1(23) 2(321) = 7383 ▼  87 7383 321 -123 42 290 -2  multiply 2(321) 1(7383) = 2369943 ▼  87 7383 2369943 -123 42 90 -2  decrease – all - 1 ▼  86 7383 2369942 -124 41 89 -3 |
| 1 2 3 4  swap 0 1  swap 1 2  swap 2 3  multiply 1 2  decrease  end | 1, 11, 3, 0 |  |

|  |  |  |
| --- | --- | --- |
| **Input JavaScript** | **Output** | **Comments** |
| [  '23 -2 321 87 42 90 -123',  'swap 1 3',  'swap 3 6',  'swap 1 0',  'multiply 1 2',  'multiply 2 1',  'decrease',  'end'  ] | 86, 7382, 2369942, -124, 41, 89, -3 | 23 -2 321 87 42 90 -123 – initial values  swap 1(-2) and 3(87) ▼  23 87 321 -2 42 90 -123  swap 3(-2) and 6(-123) ▼  23 87 321 -123 42 90 -2  swap 1(87) and 0(23) ▼  87 23 321 -123 42 90 -2  multiply 1(23) 2(321) = 7383 ▼  87 7383 321 -123 42 290 -2  multiply 2(321) 1(7383) = 2369943 ▼  87 7383 2369943 -123 42 90 -2  decrease – all - 1 ▼  86 7383 2369942 -124 41 89 -3 |
| [  '1 2 3 4',  'swap 0 1',  'swap 1 2',  'swap 2 3',  'multiply 1 2',  'decrease',  'end'  ] | 1, 11, 3, 0 |  |

# 03. Numbers

Write a program to **read a sequence of integers** and find and print the **top 5** numbers that are **greater than the average** value in the sequence, sorted in descending order.

## Input

Read from the console a single line holding space separated number.

## Output

Print the above described numbers on a single line, space separated. If **less than 5 numbers** hold the above mentioned property, print less than 5 numbers. Print “**No**” if no numbers hold the above property.

## Constraints

All input numbers are integers in range [-1 000 000 … 1 000 000]. The count of numbers is in range [1…10 000].

## Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 10 20 30 40 50 | 50 40 | Average number = 30.  Numbers greater than 30 are: {40, 50}.  The top 5 numbers among them in descending order are: {50, 40}.  Note that we have only 2 numbers, so all of them are included in the top 5. |
| 5 2 3 4 -10 30 40 50 20 50 60 60 51 | 60 60 51 50 50 | Average number = 28.08.  Numbers greater than 20.078 are: {30, 40, 50, 50, 60, 60, 51}.  The top 5 numbers among them in descending order are: {60, 60, 51, 50, 50}. |
| 1 | No | Average number = 1.  There are no numbers, greater than 1. |
| -1 -2 -3 -4 -5 -6 | -1 -2 -3 | Average number = -3.5.  Numbers greater than -3.5 are: {-1, -2, -3}.  The top 5 numbers among them in descending order are: {-1, -2, -3}. |

|  |  |  |
| --- | --- | --- |
| **Input JavaScript** | **Output** | **Comments** |
| '10 20 30 40 50' | 50 40 | Average number = 30.  Numbers greater than 30 are: {40, 50}.  The top 5 numbers among them in descending order are: {50, 40}.  Note that we have only 2 numbers, so all of them are included in the top 5. |
| '5 2 3 4 -10 30 40 50 20 50 60 60 51' | 60 60 51 50 50 | Average number = 28.08.  Numbers greater than 20.078 are: {30, 40, 50, 50, 60, 60, 51}.  The top 5 numbers among them in descending order are: {60, 60, 51, 50, 50}. |
| '1' | No | Average number = 1.  There are no numbers, greater than 1. |
| '-1 -2 -3 -4 -5 -6' | -1 -2 -3 | Average number = -3.5.  Numbers greater than -3.5 are: {-1, -2, -3}.  The top 5 numbers among them in descending order are: {-1, -2, -3}. |

# Programming Fundamentals Mid Exam Retake 07 April 2020

## Problem 1. Counter Strike

Write a program that **keeps track of every won** battle against an **enemy**. You will receive **initial energy**. Afterwards you will start receiving the **distance** you need to **go to reach an enemy** until the **"End of battle"** command is given, or until you **run out of energy.**

The **energy** you need for reaching an enemy is **equal to the distance you receive**. Each time you reach an enemy, your **energy is reduced.** This is considered a successful battle (**win**). If you don't have **enough energy** to reach an the enemy, print:

**"Not enough energy! Game ends with {count} won battles and {energy} energy"**

and **end the program.**

Every **third won battle** increases **your energy with the value of your current count of won battles**.

Upon receiving the **"End of battle"** command**,** print the **count of won battles** in the following format:

### "Won battles: {count}. Energy left: {energy}"

### Input / Constraints

* On the **first line** you will receive **initial energy** – an **integer [1-10000]**.
* On the **next lines,** you will be receiving **distance** of the enemy – an **integer** **[1-10000]**

### Output

* The description contains the proper output messages for each case and the format in which they   
  should be print.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 100  10  10  10  1  2  3  73  10 | Not enough energy! Game ends with 7 won battles and 0 energy | Initial energy is 100. The first distance is 10, so we subtract 10 from 100 and we consider this a **won** battle. We are left with 90 energy. Next distance – 10, and 80 energy left.  Next distance – 10, 3 won battles and 70 energy, but since we have 3 won battles, we increase the energy with the current count of won battle, in this case – **3 and it becomes 73**.  The last distance we receive – **10** is unreachalble since we have **0** energy, so we print the appropriate message and the program ends. |
| 200  54  14  28  13  End of battle | Won battles: 4. Energy left: 94 |  |

# Programming Fundamentals Mid Exam Retake 07 April 2020

## Problem 2. Shoot for the Win

Write a program that helps you keep track of your **shot targets**. You will receive a **sequence with integers**, separated by single space, representing targets and their value. Afterwards, you will be receiving indices until the **"End"** command is given and you need to print the **targets** and the **count of shot targets**.

Every time you receive an **index**, you need to shoot the target on that index, **if it is possiblie**.

Everytime you **shoot a target**, its value becomes **-1 and it is considered shot**. Along with that you also need to:

* **Reduce** all the other **targets**, which have **greater values** than your **current** target, **with its value**.
* All the **targets**, which **have less than or equal** value to the **shot target**, you need to **increase** **with its value.**

**Keep in mind that you can't shoot a target, which is already shot.** **You also can't increase or reduce a target, which is considered shot.**

When you receive the **"End"** command, print the targets in their current state and the **count of shot targets** in the following format:

**"Shot targets: {count} -> {target1} {target2}… {targetn}"**

### Input / Constraints

* On the **first line** of input you will receive a **sequence** of **integers**, **separated** by **a single space – the targets sequence**.
* On the **next lines**, until the **"End"** command you be receiving **integers** each on a single line – **the index of the target to be shot.**

### Output

* The format of the output is described above in the problem description.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 24 50 36 70  0  4  3  1  End | Shot targets 3 -> -1 -1 130 -1 | First we shoot target on index 0. It becomes equal to -1 and we start going through the rest of the targets. Since 50 is more than 24, we reduce it to 26 and 36 to 12 and 70 to 46. The sequence looks like that:  **-1 26 12 46**  The next index is invalid, so we don't do anything. Index 3 is valid and after the operations our sequence should look like that:  **-1 72 58 -1**  Then we take the first index with value 72 and our sequence looks like that:  **-1 -1 130 -1**  Then we print the result after the **"End"** command. |
| 30 30 12 60 54 66  5  2  4  0  End | Shot targets: 4 -> -1 120 -1 66 -1 -1 |  |

# Programming Fundamentals Mid Exam Retake 07 April 2020

## Problem 3. Moving Target

You are at the shooting gallery again and you need a program that helps you keep track of moving targets. On the first line, you will receive a **sequence of targets with their integer values**, split by a **single space**. Then, you will start receiving **commands for manipulating the targets**, until the **"End"** command. The commands are the following:

* **Shoot {index} {power}**
  + Shoot the target at the index, **if it exists** by **reducing** its **value** by the **given** **power** (**integer value**).A target is considered **shot** when **its value reaches 0**.
  + Remove the target, **if it is shot**.
* **Add {index} {value}**
  + Insert a target with the received value at the received **index, if it exist**. If not, print: **"Invalid placement!"**
* **Strike {index} {radius}**
  + Remove the **target at the given index** and the **ones before and after it depending on the radius, if such exist. If any of the indices in the range is invalid print:**

**"Strike missed!" and skip this command.**

**Example:** **Strike 2 2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | {radius} | {radius} | {strikeIndex} | {radius} | {radius} |  |  |

* **End**
  + Print the sequence with targets in the following format:

**{target1}|{target2}…|{targetn}**

### Input / Constraints

* On the **first line** you will receive **the sequence of targets** – **integer values [1-10000]**.
* On the **next lines,** until the **"End"** will be receiving the command described above – **strings**.
* There will never be a case when **"Strike"** command would empty the whole sequence.

### Output

* Print the appropriate message in case of **"Strike"** command if necessary.
* In the end, print the sequence of targets in the format described above.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 52 74 23 44 96 110  Shoot 5 10  Shoot 1 80  Strike 2 1  Add 22 3  End | Invalid placement!  52|100 | The first command is "**Shoot**", so we reduce the target on **index** **5**, which is valid, with the given **power** – **10**.  Then we receive the same command but we need to reduce the target on the 1st index, with power 80. The value of this target is 74, so it is considered shot and we **remove** it.  Then we receive the "**Strike**" command on the 2nd index and we need to check if the range with radius 1 is valid:  **52 23 44 96 100**  And it is, so we **remove** the targets.  At last we receive the "**Add**" command, but the index is **invalid** so we print the appropriate **message** and in the end we have the following result:  **52|100** |
| 1 2 3 4 5  Strike 0 1  End | Strike missed!  1|2|3|4|5 |  |

# Programming Fundamentals Final Exam 04.04.2020

## Problem 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 |

# Programming Fundamentals Final Exam 04.04.2020

# Problem 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. | |

# Problem 3. Heart Delivery

*Valentine’s Day is coming, and Cupid has very limited time to spread some love across the neighborhood. Help him with his mission!*

You will receive a **string** with **even integers,** separated by a **"@".** This is our neighborhood. After that a series of **Jump** commands will follow, until you receive **"Love!"** Every house in the neighborhood needs a certain number of **hearts** delivered by Cupid, in order to be able to celebrate Valentine’s Day. Those needed hearts are indicated by the integers in the neighborhood.

Cupid starts at the position of the **first** **house** (index 0) and must jump by a **given length.** The jump commands will be in this format: **"Jump {length}"**.

Every time he jumps from one house to another, the needed hearts for the visited house are **decreased by 2**. If the needed hearts for a certain house become **equal to 0** , print on the console **"Place {houseIndex} has Valentine's day."** If **Cupid** jumps to a house where the needed hearts are **already** **0,** print on the console"**Place {houseIndex} already had Valentine's day.**".

Keep in mind that **Cupid** can have a **bigger jump length** than the **size of the neighborhood** and if he does jump **outside** of it, he should **start** from the **first house** again**.**

*For example, we are given this neighborhood: 6@6@6. Cupid is at the start and jumps with a length of 2. He will end up at index 2 and decrease the needed hearts there by 2: [6, 6, 4]. Next he jumps again with a length of 2 and goes outside the neighborhood, so he goes back to the first house (index 0) and again decreases the needed hearts there: [4, 6, 4].*

### Input

* On the first line you will receive a **string** with **even integers** separated by **"@"** –the neighborhood and the number of hearts for each house.
* On the next lines, until "**Love!**" is received, you will be getting jump commands in this format: "**Jump {length}**".

### Output

At the end print **Cupid's** **last position** and whether his mission was successful or not:

* "**Cupid's last position was {lastPositionIndex}.**"
* If **each house** has had a Valentine's day, print:
  + "**Mission was successful.**"
* If **not,** print the **count** of all houses that **didn`t** celebrate a Valentine's Day:
  + **"Cupid has failed {houseCount} places."**

### Constraints

* The **neighborhood`s** size will be in the range [1…20]
* Each **house** will need an **even number** of hearts in the range [2 … 10]
* Each **jump length** will be an integer in the range [1 … 20]

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 10@10@10@2  Jump 1  Jump 2  Love! | Place 3 has Valentine's day.  Cupid's last position was 3.  Cupid has failed 3 places. | Jump 1 ->> [10, 8, 10, 2]  Jump 2 ->> [10, 8, 10, 0] so we print "Place 3 has Valentine's day."  Next command is "Love!", so we print Cupid`s last position and the outcome of his mission. |
| 2@4@2  Jump 2  Jump 2  Jump 8  Jump 3  Jump 1  Love! | Place 2 has Valentine's day.  Place 0 has Valentine's day.  Place 0 already had Valentine's day.  Place 0 already had Valentine's day.  Cupid's last position was 1.  Cupid has failed 1 places. |  |

# Programming Fundamentals Final Exam 04.04.2020

## Problem 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 |

# Problem 2. Shopping List

*It’s the end of the week and it is time for you to go shopping, so you need to create a shopping list first.*

### Input

You will receive an **initial list** with groceries separated by **"!"**.

After that you will be receiving **4 types** of commands, until you receive **"Go Shopping!"**

* **Urgent {item} -** **add** the item at the **start** of the list. If the item **already exists,** skip this command.
* **Unnecessary {item} - remove** the item with the given name, only **if it exists** in the list. Otherwise skip this command.
* **Correct {oldItem} {newItem} –** if the item with the given **old name** exists, **change** its name with the **new** one. If it **doesn't exist**, skip this command.
* **Rearrange {item} -** if the grocery exists in the list, **remove** it from its **current position** and **add** it at the **end** of the list.

### Constraints

* There won`t be any duplicate items in the initial list

### Output

Print the **list** with all the groceries, joined by **", ".**

* **"{firstGrocery}, {secondGrocery}, …{nthGrocery}"**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Tomatoes!Potatoes!Bread  Unnecessary Milk  Urgent Tomatoes  Go Shopping! | Tomatoes, Potatoes, Bread |
| **Input** | **Output** |
| Milk!Pepper!Salt!Water!Banana  Urgent Salt  Unnecessary Grapes  Correct Pepper Onion  Rearrange Grapes  Correct Tomatoes Potatoes  Go Shopping! | Milk, Onion, Salt, Water, Banana |

# Problem 1. National Court

*Every day thousands of people pass by the reception at "National Court" with various questions to ask and the employees have to help everyone by providing correct information and to answer all questions.*

There are **3 employees** working on the reception all day long. Each of them can handle different number of **people** **per hour**. Your task is to calculate **how much time** it will take **to** **answer all the questions** of a given number **of people**.

First you will receive 3 lines with integers, representing the **count of people** that each of the **employee can help per hour.** On the next line you will receive the **total** **people count** as a single integer.

Every **fourth hour** all the employees **have a one-hour break** before they start working again. This is the only break they get because they don`t need rest and have no personal life. Calculate the time needed to answer all people`s questions and print it in the following format: "Time needed: {time}h."

### Input / Constraints

* On first three lines - **each employee`s efficiency** - an integer in the range **[1 - 100]**
* On the fourth line - **people count** – an integer in the range **[0 – 10000]**
* Input will always be valid and in the range specified

### Output

* Print a single line: "Time needed: {time}h."
* Allowed working **time** / **memory**: **100ms** / **16MB**

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 5  6  4  20 | Time needed: 2h. | All employees can answer 15 people per hour. After the first hour there are 5 people left to be answered.  All people will be answered in the second hour. |
| 1  2  3  45 | Time needed: 10h. | All employess can answer 6 people per hour. In the first 3 hours they have answered 6 \* 3 = 18 people. Then they have a break for an hour.  After the next 3 hours there are  18 + 6 \* 3 = 36 answered people.  After the break for an hour, there are only 9 people to answer.  So in the 10th hour all of the people questions would be answered. |
| 3  2  5  40 | Time needed: 5h. |  |