## Cure the virus | 3 point(s)

There is an outbreak of VirusX in Redmond. The local authorities have discovered a vaccine that protects against infection by VirusX. However, they can only produce so many vaccines per week. Your job is to figure out **the best order in which people should be vaccinated**. Everyone will get vaccinated, but the people more likely to be sick should be vaccinated first. You are more likely to get sick if the virus` DNA closely matches somewhere in your DNA.

However, there is a catch: **the virus can mutate**. Over time, the virus can “cut” letters from its DNA. So it is possible that theAAAAG virus mutates into the AAAG, AAAA, …, A viruses. Removing letters is the only possible mutation: the virus cannot **change**any letter.

For example, if your DNA is GGGGGGGAAAAGGGGGG and the virus DNA is AAAAG, you have 1.0 probability of getting infected because all of the letters of the virus are found in the same order in your DNA. If your DNA was TTTTTTTAAAATTTTTT, then the virus only partially matches your DNA. For partial matches, we define the probability to get sick by the numbers of continuous letters of the virus’ DNA that matches your DNA divided by the total number of letters in the un-mutated virus’ DNA. With the example above, AAAA is the longest possible mutation that matches in your DNA, so your probability of getting sick is len(AAAA) / len(AAAAG): 0.8.

You are provided with the DNA of every inhabitant of Redmond and the virus’ DNA. For every inhabitant, you need to output the order in which he/she should get vaccinated. Everyone that has the same probability to get sick should get vaccinated in the same batch. For example, the order of vaccination for inhabitants with the following probability to get sick: [0.0, 0.5, 0.5, 0.0, 1.0] is: [3, 2, 2, 3, 1].

### Tip

Here is a tip for how to get from the “Canonical Sample Input” to “Canonical Sample Output” (described below). Given the TAATvirus and its highest infection score possible mutation for persons 0 to 4, here are the longest matching mutations and the attached probability to get sick for every person:

Person #0: T, 0.250 Person #1: TAA, 0.750 Person #2: TT, 0.500 Person #3: TAAT, 1.000 Person #4: TA, 0.500

Therefore, person 3 has a 1.0 probability of getting sick, so he or she gets vaccinated first. Person 1 is the next highest, so he or she gets vaccinated second. We continue to vaccinate in batches of probability of getting sick, with person 0 getting vaccinated last because he or she has the least probability to get sick.

Note that what is described above is not the output you should generate, but a tip on how you can get to the desired output inside your code.

### Input description/format

The first line of the file contains three numbers: the number of citizens in Redmond, the length of the citizen’s DNA and the length of the virus’ DNA.

This is followed by one sequence of DNA per line, representing in order the citizens’ DNA. The last DNA sequence of the file is the virus’ DNA.

You can assume that the virus’ DNA sequence is always shorter than the citizen’s DNA sequences.

### Output description/format

The below example becomes clear if we give the “DNA match score” as well as the longest sequence of DNA matched.

Person #0: G, 0.333   
Person #1: GCC, 1.000   
Person #2: CC, 0.667   
Person #3: GC, 0.667   
Person #4: GC, 0.667

Person 1 has a 1.0 probability of getting sick, so he gets vaccinated first. Persons 2, 3, and 4 are the next highest probable to get sick, so they get vaccinated second. Person 0 has the least chance to get sick; he gets vaccinated last.

### Example input

5 10 3 AGGAAAAGAG CGAGGCCAAC GACAAAACCG GCGACGCAGA AACAGCGCAG GCC

### Example output

Person #0: 3. Person #1: 1. Person #2: 2. Person #3: 2. Person #4: 2.

## OData service | 2 point(s)

The Open Data Protocol (OData) is a data access protocol created by Microsoft that builds on simple REST principles. Your task is to retrieve certain data from a sample (live) OData service.

For this problem you will be working with the OData service located [here](http://services.odata.org/Northwind/Northwind.svc/" \t "_blank) ([http://services.odata.org/Northwind/Northwind.svc/](http://services.odata.org/Northwind/Northwind.svc/" \t "_blank)). You will retrieve data from an entity set called ‘Products’. An entity set is equivalent to a table in a relational database. The Products entity set is located [here](http://services.odata.org/Northwind/Northwind.svc/Products" \t "_blank) ([http://services.odata.org/Northwind/Northwind.svc/Products](http://services.odata.org/Northwind/Northwind.svc/Products" \t "_blank)).

Your input file will consist of a series of simple queries to perform on the Products entity set. The input will contain a $filterexpression to perform on Products. This is equivalent to a WHERE clause in SQL. Your job is simply to see how many entities (rows in a table) are returned from the specified query.

Normally a request to the endpoint specified above would return product data - but since you are only interested in the number of entities that would be returned, you can send requests to this special $count [endpoint](http://services.odata.org/Northwind/Northwind.svc/Products/%24count" \t "_blank)   
([http://services.odata.org/Northwind/Northwind.svc/Products/$count](http://services.odata.org/Northwind/Northwind.svc/Products/$count" \t "_blank)). After sending an appropriate request to this endpoint, it will respond with the number of entities that match the query. Print out this value on a line in the output file. Your output file will have one line per line of input containing one number.

### Input description/format

The input will be a series of $filter queries to perform. Each line of input will correspond to a single query.

For instance, the input line:   
Discontinued eq true

should send a GET request to the following URL:http://services.odata.org/Northwind/Northwind.svc/Products/$count?$filter=Discontinued eq true

### Output description/format

Your output will be a single number per line of input, indicating how many entities match the given query.

### Example input

Discontinued eq true SupplierID eq 4 CategoryID eq 2

### Example output

8 3 12

## Unicode email addresses | 2 point(s)

With the expansion of the Internet across the world, the demand is growing for everyone to have an email address. Until everyone is ready to use international characters, you are now in charge of writing a converter that will take UTF-8 characters and bring them back to something DNS servers will be able to handle.

To do so, you must take an email string and break it into 3 pieces: first@second.third. Process each piece as follows:

1. Leave all characters that have ASCII codes between 33 and 126 as is, except convert uppercase letters to lowercase letters
2. Represent all other characters (referred to hereinafter as special characters) using the format +position?UTF-8\_hexadecimal\_representation
   * The position is the 0-indexed position of the represented character
   * The UTF-8 hexadecimal representation must not contain spaces and must be finalized

If at least one special character was present in the string, append a ~ and each special character at the end of the address.

For example, the piece Pøpø would be processed to pp+1?C3B8+3?C3B8~øø

## Clarifications

* There shouldn’t be any spaces.
* You will receive only characters contained within the UTF-8 standard

### Input description/format

Each line will be an email address with Unicode characters.

### Output description/format

<new email address composition>~<characters removed in order>

### Example input

WoRlD\_ReAdY@Microsoft.COM Café@Chaud.tôt Trinitŷ@߷.com ա@մ.կ 😎@🚀.com ep@ез.кom Soup@🍚.com 🆍🆌@SAPA.com ಠಠ@Look.com ᱪ@Alien.invasion Vikingoftheᛗ@North.ᛣ ဩ@ဦ.com Ⴭaჯ@Greorgian.letTer

### Example output

world\_ready@microsoft.com caf+3?C3A9@chaud.tt+1?C3B4~éô trinit+6?C5B7@+0?DFB7.com~ŷ߷ +0?D5A1@+1?D5B4.+0?D5AF~ամկ +0?EFBFBD+1?EFBFBD@+1?EFBFBD+2?EFBFBD.com~😎🚀 ep@+0?D0B5+1?D0B7.om+0?D0BA~езк soup@+0?EFBFBD+1?EFBFBD.com~🍚 +0?EFBFBD+1?EFBFBD+2?EFBFBD+3?EFBFBD@sapa.com~🆍🆌 +0?E0B2A0+1?E0B2A0@look.com~ಠಠ +0?E1B1AA@alien.invasion~ᱪ vikingofthe+14?E19B97@north.+0?E19BA3~ᛗᛣ +0?E180A9@+0?E180A6.com~ဩဦ a+0?E1838D+2?E183AF@greorgian.letter~Ⴭჯ

## Vending machine | 2 point(s)

You are to design a software driver for a vending machine. The vending machine operator will provide a list of inventory. A list of user inputs (change deposited, buttons pressed) will also be provided.

Items are referenced by a number and a letter (e.g. A4, E8). This will always be a single letter and a single number.

## Actions

Buttons and change may be pressed and deposited in any order. Once a valid combination of total change deposited and buttons pressed has been reached, the machine will dispense the corresponding item. The refund button (represented by a # character) will clear any previously pressed buttons. Invalid selections will reset the machine’s recorded button presses; the deposited change will remain.

### Example Actions

Assume A3 costs 0.95.

* Actions,A,3,$1 (Dispense A3)
* Actions,A,3,$0.5,$0.5 (Dispense A3)
* Actions,$0.5,A,$0.5,3 (Dispense A3)
* Actions,A,3,$0.5,#,$0.5,$0.5 (Nothing dispensed yet, but there is $1 in the machine)

### Invalid Selections

* 2 letters in a row.
* 2 numbers in a row.
* 1 letter and 1 number for a slot that wasn’t in the original inventory.
* 1 letter and 1 number for a slot that is now empty.
* 1 letter or 1 number after a valid selection was obtained.

## Change

Change is given from the initial change provided as a part of the inventory list. Change is represented by a $ character followed by a slot number (single digit, 0-9). Change is dispensed using a greedy algorithm. If we are unable to give exact change (the machine ran out of a particular small denomination coin), the next smallest denomination will be returned in its place; in other words, the customer will never be overcharged, but they may end up paying less for the item. It is noted that the customer may also end up paying less if the greedy algorithm fails to yield exact change.

## Output

A final inventory must be the output of this program; this will include the final counts of all product.   
This inventory output will be sorted by their item id (“A4”, etc.) in alphabetical order with the change slots listed after the inventory items.   
The total present in the deposit box will also be provided on the last line of the file prefaced by a pair of $ symbols. Total deposited is expressed with two digits after the decimal place. Any cash that was deposited but not refunded or used to purchase anything at the end of all actions will be added to the deposit box total.

### Input description/format

The input file consists of a list of inventory to be initially stocked and a series of Actions taken.

The item inventory rows take the form id,name,cost,quantity, where:

* id: letter and a number (upper-case letters)
* name: string representing the item’s name (case sensitive)
* cost: the dollar amount of the item
* quantity: number of the particular item loaded into the machine

A1,MarsBar,0.05,1   
B1,KitKat,0.65,12

The change inventory rows take the form changeslot,value,quantity, where:

* changeslot: $ followed by a 0-indexed single digit (so, there are a max of 10 slots)
* value: numerical value of this change, to 2 digits after the decimal place in all cases
* quantity: number of units of change available

$1,0.05,100   
$2,0.25,200

After the change inventory, the last row will be the actions taken on the machine. This takes the form Actions,...actions.... Action Values can be the following:

* A single letter (case insensitive) or number is considered to be a button press
* Anything following a $ character is deposited change
* The # character represents a refund request.

Actions,$1,A,2,#,B,3,$0.05   
or   
Actions,$0.05,$0.05,$0.05

All input will be formatted correctly, no invalid inputs will be provided.   
Odd series of actions (such as repeated refund requests or dispensing empty slots should, however, be handled gracefully).

### Output description/format

The output will be a list of the remaining inventory (items and change) followed by a total of the money present in the deposit box.

The items are listed alphabetically by their item id, followed by the change slots, followed by the deposit box total represented by a$$ symbol.

Item output takes the form id,name,quantity:

* id: Same format as input, uppercase letter and a number
* name: Identical to the input provided name, case sensitive
* quantity: The remaining count of this item

A1,Candy1,0   
B1,Candy7,5

Changeslot output takes the form changeslot,quantity:

* changeslot: $ character and a number, same as the input
* quantity: remaining inventory of change

$1,82   
$2,190

Deposit box total takes the form $$,total:

* total: value of the deposited cash (2 decimal places)

$$,8.00

### Example input

A1,Candy1,0.05,1 B1,Candy7,0.65,12 $1,0.05,100 $2,0.25,200 Actions,$1,A,1,$1,B,1,$1,B,1,$1,B,1,$1,B,1,$1,B,1,$1,B,1,$1,B,1,$1,B,1

### Example output

A1,Candy1,0 B1,Candy7,4 $1,80 $2,189 $$,9.00

## A variation on anagrams | 1 point(s)

Let's assume you were recently hired by a Start-up. On the first day, you meet the CEO and are given an assignment. Your CEO has decided that all products will be named according to the following pattern.

Your CEO calls the pattern “P@tter”. Two product names obey P@tter if they are permutations of each other, ignoring spaces and capitalization.

Your task is to determine whether two product names obey P@tter.

### Input description/format

The input will consist of two product names per line, in quotation marks, which are not part of the names. The quoted names will be separated by a comma and potentially whitespace.  
You can assume that there will be ≤ 1000 lines in the input file. Also, product name length will be ≤ 255 characters and product names will contain only ASCII characters.

### Output description/format

For each line of input, output the result in one of the following 2 ways (case sensitive):

* Valid Pattern
* Invalid Pattern

### Example input

"Calculate", "Acute Call" "Drop Cue" , "Cued Pro" "carE Not", "raCe On"

### Example output

Valid Pattern Valid Pattern Invalid Pattern

Detecting unreachable instructions | 1 point(s)

Untouched instructions in code can cause some serious problems. You may have heard about a recent security hole related to [this topic](https://www.imperialviolet.org/2014/02/22/applebug.html" \t "_blank).

Your task is to write a small program that catches the “untouched” lines of code written in our simplified programming language.

The program starts at line 1. You may assume the program will be terminated if it infinitely loops or continues past its last instruction.

### Input description/format

Each line of input contains an instruction (NEXT or GOTO).

1. NEXT: the program will move to the next line.
2. GOTO X: the program will go to line number X.

### Output description/format

Please print out the line number of instructions that the program doesn't touch (one line number per line in the output file).

### Example input

NEXT NEXT GOTO 5 NEXT GOTO 1 NEXT

### Example output

4 6