C++ Fundamentals: Exam

The following tasks should be submitted to the SoftUni Judge system, which will be open starting Sunday, 14 January 2018, 09:00 (in the morning) and will close the same day at 15:00. Submit your solutions here: https://judge.softuni.bg/Contests/Compete/Index/907.

For this exam, the code for each task should be a single C++ file, the contents of which you copy-paste into the Judge system.

Please be mindful of the strict input and output requirements for each task, as well as any additional requirements on running time, used memory, etc., as the tasks are evaluated automatically and not following the requirements strictly may result in your program's output being evaluated as incorrect, even if the program's logic is mostly correct.

You can use C++03 and C++11 features in your code.

Unless explicitly stated, any integer input fits into **`int`** and any floating-point input can be stored in **`double`**. On the Judge system, a C++ **`int`** is a **`32-bit`** signed integer and a C++ **`double`** is a **`64-bit`** IEEE754 floating point number.

NOTE: the tasks here are NOT ordered by difficulty level.

Task 3 – Data (Exam-Task-3-Data)

Non-relational databases (frequently called NoSQL, or schema-less) keep entities (records) as collections of key-value pairs, instead of having a strict format for the entities. That means that there are no strict "types" of entities and each entity can have different key-value pairs. Each entity can have different properties/fields.

An example of such a datastore, containing cities, might contain the following entities (note that each line here will be an entity, described by pairs – every first item in a pair is the name of the field, every second item in a pair is the value of the field):

```
name Gabrovo population 66551 cats many catswithtails none
name Townsville country USA population 1214
powerpuffgirls 3 name Townsville country fictionalUSA
population 186757 name Townsville country Australia
name Jadotville country Congo description TheIrishDontLikeIt
place UnnamedCity description itsACityInAGameAboutConanIThink population WhoKnows
```

(yep, there are multiple Townsvilles in the world, who would have thought… But Gabrovo is unique!... unless you count asteroids)

To be able to quickly operate on entities, such databases are usually "**indexed**" by one of the fields of some of the entities (there are usually fields which most entities have, but that's not always guaranteed). In our example with the cities, such a database might be indexed by the **`name`** field.

Indexing items by a field allows quick access to items having that field – the index allows fast searches for a value of the index. In our cities example, having an index on city names would mean that we can quickly search for the city "**`Gabrovo`**" or the city "**`Jadotville`**" and access it's other fields. If we search for the city "Townsville", we'd get 3 results.

For example, we could ask the database "***show me the populations of Townsville cities***" like this:

```
Townsville population
```

And we'd get the following result (since there are multiple results, we get them in the order of input):

```
1214 186757
```

Note that there are **3 `Townsville`**s but only **2** results – that's because the second **`Townsville`** doesn't have a **`population`** field.

Your task is to write a program which reads an index field (the name of the field), a sequence of entities (where each entity is a collection of key-value pairs representing its fields and entities can have different fields), and answers a sequence of queries by the index, asking for the values of the fields of the entities found by the index.

Queries will have an **index value** and the **name of the field**, for which you should print the contents. For each query:

- Find all entities which have an **index value** matching the query
- For each found entity, if it has a **field** matching the **name of the field**, print the **contents of that field** (if it doesn't, don't print anything). The contents should be separated by **single spaces** (be careful not to have a double-space when one of the entities does not contain the field we are looking for). If no entity has a field with the **name of field** we are searching for, print an **empty line**.
- If there are no entities which have an **index value** matching the query, print **`[not found]`**

Input

The input will be read from the console (i.e. read from the standard input).

The input will consist of **2 "parts"** – the **entities part** and the **queries part**. The entities part will end when a line containing the string **`[queries]`** is entered. That line will signal the beginning of the queries and all remaining lines will contain queries which the program should answer, until a line containing the string **end** is entered. That line will end the program.

Each line in the **entities part** will contain the information about a single entity. That information will be in the form of an **even** number of **strings, separated by single spaces**. Each string can contain English letters and numbers (a-z, A-Z, 0-9). These strings will represent the **field names** and **values** of the entity. If we number the strings from 0 to N-1 (where N is the number of strings on the line), then **`string 0`** is the **name** of the **first** field and **`string 1`** is the **value** of that field, **`string 2`** is the **name** of the **second** field and **`string 3`** the **value** of the **second** field and so on.

Each line in the queries part will contain exactly two strings (sequences of a-z, A-Z, 0-9), separated by a single space – the **index value** we are searching for and the **name of the field**, for which we need to print the contents.

Output

For **each query**, print a **single line**, containing the **contents of the fields for which the query asks**, taken from the **entities which match the index being searched for**, separated by **single spaces**.

If one of the found entities does not contain the field for which the query asks, do not print anything for that entity.

If no entity contains the field, print an empty line.

If no entities match the index being searched for, print **`[not found]`**.

Restrictions

There will be no more than **`300`** entities in the input.

There will be no more than **`300`** queries.

The total running time of your program should be no more than **`0.1s`**

The total memory allowed for use by your program is **16MB**

Example I/O

Example Input	Expected Output	Explanation
age name Joro age 25 place unknown age 25 name Ina place Sofia weight 13 name Dog weight 5 name Cat age 25 place SomeCage age 25 weight 1 name Canary [queries] 25 place 25 name 26 name end	unknown Sofia SomeCage Joro Ina Cat Canary [not found]	The first query searches for the "place" of all entities with an "age" of 25. All entities except Dog have that age, but the Cat does not have a place, so we don't print anything for it. In the order of input, the result is "unknown" for Joro, "Sofia" for Ina and "SomeCage" for the Canary. The second search is

		again by the same age, but asks for a name – we match the same entities as the previous query, but this time all of them have a name and we print them all. The last query does not find any entity with an age of 26, so we print [not found]. Note: if we had a query “25 future”, we would get an empty line – there are entities with age==25, but none of them has a “future” (... that was dark)