Data Types

- **ID/password strings**: either is of length $\in \{1, 2, \ldots, 100\}$ and is formed only by characters $\in \{0, \ldots, 9, A, \ldots, Z, a, \ldots, z\}$
- **money in account**: integer within $[0, 10^{10}]$

Rules

1 Password Encoding

Security is a serious issue for all account systems, especially those in banks. Storing the passwords as plain text in the system is known as a less safe way as someone who gains internal access to the system can easily obtain the password and use them to do further damage.

Therefore, in this final project, we ask you to store a **hashed** password in your system (`http://en.wikipedia.org/wiki/Password`). We ask you to check the MD5 message-digest algorithm to encode your passwords, and you are asked to store the **hashed** passwords rather than plain text in your system. The C++ compatible code for calculating MD5 hash code of a string can be found here: `http://www.zedwood.com/article/cpp-md5-function`.

2 Score of IDs

In operation `create`, suppose the account ID already exists, you should recommend 10 best account IDs from all unused IDs; in operation `transfer`, suppose the account ID cannot be found, you should recommend 10 best account IDs from all existing IDs. In both operations, the order of your outputs should follow a `score` function to be specified below, in ascending order. For outputs with the same `score`, please follow the ascending dictionary order with the alphabets $\{0, \ldots, 9, A, \ldots, Z, a, \ldots, z\}$ (using ASCII values for alphabet-level comparison).

Consider two account IDs u and v, where the index i of strings starts from 0. `score`(u, v) is defined as follows.

$$\texttt{score}(u, v) = \sum_{i=1}^{\Delta L} i + \sum_{i=0}^{L-1} \big((u[i] \neq v[i]) \ ? \ (L - i) \ : \ 0\big),$$

where `?:` corresponds to the ternary conditional operator in C, $L = \min(\texttt{strlen}(u), \texttt{strlen}(v))$, and $\Delta L = |\texttt{strlen}(u) - \texttt{strlen}(v)|$. Some examples are as follows.

`score("abc", "abcd")` $= 1 + 0 = 1$

`score("abce", "abcd")` $= 0 + 1 = 1$

`score("abed", "abcd")` $= 0 + 2 = 2$

`score("abde", "abcd")` $= 0 + (2 + 1) = 3$

`score("abcefg", "abcd")` $= (1 + 2) + 1 = 4$

So if `"abcd"` is the account being considered in `create` or `transfer`, and those strings are among the list to be outputted, the output order is `"abc"`, `"abce"`, `"abed"`, `"abde"`, `"abcefg"`.

Input/Output Format

You program will read from the standard input, process a sequence of operations, and output to the standard output. The following are the operations you need to handle. Notice that texts after // (including // itself) are explanations of that output, and you do not need to and should not print those.

The operations can be separated into two types. The first type contains operations that that are not bound to the last successful login. They should be handled without considering the last-logged-in user. The second type contains operations that are bound to the last successful login. You can assume that there will be at least one successful login operation before any of the second-type operations. Since the second type of operations are bound to the last successful login, it is *as if* the system serves one active user only. That is, when a new user performs a successful login, it is *as if* the original user is forced to logout.

Operations that are not bound to last successful login

- `login`: login to a specific account
  - **Input:** `login [ID] [password]`
  - **Output:**

    `ID [ID] not found` *//no such account ID*

    `wrong password` *//wrong password*

    `success` *//successful login operation*
  - **Note:** If there are multiple matching cases above, such as "ID [ID] not found" and "wrong password", please output only the first matching case, which is "ID [ID] not found." That is, you can basically treat the multiple cases in **Output** as `if ... else if ... else if ...`. All outputs of the operations below follow this rule.
- `create`: create a specific account with password
  - **Input:** `create [ID] [password]`
  - **Output:**

    `ID [ID] exists, [best 10 unused IDs (separated by ",")]` *//ID already exists*

    `success` *//successful create operation*
  - **Note:** If there are fewer than 10 unused account IDs, just output them all.
- `delete`: delete a specific account with password
  - **Input:** `delete [ID] [password]`
  - **Output:**

    `ID [ID] not found` *//no such account ID*

    `wrong password` *//wrong password*

    `success` *//successful delete operation*
  - **Note:** You can assume that `[ID]` $\neq$ `[last-successful-login-ID]`.
- `merge`: merge the second account into the first one; after merging, the second one is deleted
  - **Input:** `merge [ID1] [password1] [ID2] [password2]`
  - **Output:**

    `ID [ID1] not found` *//no such account ID*

    `ID [ID2] not found` *//no such account ID*

    `wrong password1` *//wrong password1*

    `wrong password2` *//wrong password2*

    `success, [ID1] has [X] dollars` *//successful, X = money of [ID1] after operation*
  - **Note:** You can assume that `[ID1]` $\neq$ `[ID2]` and `[ID2]` $\neq$ `[last-successful-login-ID]`. You need to merge two things: the money, and the **transfer history** (as if all transfer records of `[ID2]` from/to other IDs are replaced by `[ID1]`).
  - **Note note:** If `[ID1]` and `[ID2]` have transfer history between them (i.e. there will be two records at the same time stamp), the record of `[ID1]` is considered "earlier" and the other record is considered "later" during `search`

Operations that are bound to last successful login

- `deposit`: deposit money into `[last-successful-login-ID]`
  - **Input:** `deposit [num]`
  - **Output:**

    `success, [X] dollars in current account` *//X = money in account after operation*
  - **Note:** You can assume that `[X]` would be no more than 10^{10}.

- `withdraw`: withdraw money from `[last-successful-login-ID]`
  - **Input:** `withdraw [num]`
  - **Output:**

    `fail, [X] dollars only in current account`*//not enough money*

    `success, [X] dollars left in current account`*//X = money left in account*

- `transfer`: transfer money from `[last-successful-login-ID]` to a specific account
  - **Input:** `transfer [ID] [num]`
  - **Output:**

    `ID [ID] not found, [best 10 existing IDs (separated by ",")]` *//ID not found*

    `fail, [X] dollars only in current account`*//not enough money*

    `success, [X] dollars left in current account`*//X = money left in account*
  - **Note:** If there are fewer than 10 existing account IDs, just output them all; you can assume that `[ID]` ≠ `[last-successful-login-ID]`.

- `find`: find all existing account IDs that matches `[wildcard_ID]` but is different from `[last-successful-login-ID]`
  - **Input:** `find [wildcard_ID]`
  - **Output:**

    `[all satisfying IDs (separated by ",") in ascending dictionary order]`
  - **Note:** `[wildcard_ID]` is a ID string that may also contain `*` or `?`, where `*` stands for 0 or more characters, and `?` stands for exactly 1 character. There might be multiple `?` or `*`, but a `*` will not sit beside `?` nor `*`. That is, there is no `?*`, `**`, nor `*?`. The following are some examples:

    `a*b?: ab0,a0b1,a00b2,abbbbb ......`

    `a??b*: abbb,a01bbbb,a01bacb ......`

    `a*b*: ab,abb,abbbb,acccbccab ......`

    `*a?b*: aab,cccasbbb,twaiblwe ......`

    `a**b`: **will not be in the test data**

    Note that you can simply use `strcmp` in C for what we mean by "ascending dictionary order." If `strcmp(`u`, `v`)` < 0, then u should be placed earlier than v.

- `search`: search all transfer history of `[last-successful-login-ID]` from/to a specific account
  - **Input:** `search [ID]`
  - **Output:**

    ~~`ID [ID] not found` //no such account ID~~

    `no record` *//no record exists*

    `[transfer history from/to ID, line by line]` *//successful search*
  - **Note:** The order of the transfer history should be in time order, the earliest being the first. Each line should follow the format: `[From/To] [ID] [num]`. The following are some examples:

    ```
    From Frank 10000
    To Frank 2000
    ......
    ```

Sample Input/Output

Sample input

```
create g1 dsa2015

create ab0 dsa1337

create aabb dsa1126

create g1 dsa2115

login g1 dsa2015

deposit 5000

find a*b*

transfer aabb 2499

search aabb
```

Sample output

(4th line corrected)

```
success

success

success

ID g1 exists, g,g0,g10,g11,g12,g13,g14,g15,g16,g17

success

success, 5000 dollars in current account

aabb,ab0

success, 2501 dollars left in current account

To aabb 2499
```