Mohammad: I can't continue. I give up!
Arash: Why?
Mohammad: cause I have read the obfuscated source-code of Unix BASH several times, and the more I read the less I understand. Thats why I want to give up porting it to our own OS, 3KP.
Arash: Ok, Ok, just calm down. Let me think about it.

... After several minutes of thinking ...

Arash: Ok, why don't we write our own BASH! Trust me! We just need a small subset of the functionalities in the BASH. So, lets write our own version.
Mohammad: No! No! Youre kidding!

Poor Mohammad! He has a lot of tasks to do. Besides a pile of books to study for his exams, he should also make some problems for the local programming contest of their university. Considering these tasks, Mohammad wonders whether he is able to accomplish this new task ...

OK! Mohammad has found a solution. In fact, he has decided to out-source the 3KP-BASH project. In other words, he would like to hire you as a professional programmer to write 3KP-BASH for him.

Problem Definition

The BASH environment for 3KP is intended to support only eight commands. The other functionalities are decided to be available as stand-alone executable programs that can be run from our BASH (Mohammad and Arash have already written the program execution codes, so you only have to add support for these eight commands).

Before describing the supported commands, lets have a brief review of some basic things about 3KP operating system.

- **File System**
  - **File**: file is a simple object in this system. Each file has a name which is actually a string consisting of English alphabetic characters, numeric characters, and '`.`' (dot) character. A filename cannot contain two consecutive '`.`' characters. Also, the user cannot create a file or directory with its name equal to a single '`.`'. A filename is no longer than 255 characters. Note that filenames are case-sensitive. Each file has a size which is the amount of data in it. The maximum allowed size for a file is 2^{63}. A file may also have some attributes. In the first version of our BASH, we only need to know two of them which are `directory` and `hidden` attributes.
  - **Directory:** is simply a file with `directory` attribute and size of zero. In fact, a directory is used as a logical place for keeping files. In other words, each directory may contain an arbitrary number of files (or directories) inside it. An empty file system is assumed to have only one directory which is called the Root Directory.
- **Current Directory**

  At any time, there is a single directory which is assumed to be the default directory for programs to use. This directory is called the Current Directory. When our BASH starts, it sets the current directory to the root directory. A user may change the current directory during his works using the facilities supplied by BASH.
- **File and Directory Addressing**

  As you may know, there are many commands and programs which take a file or directory as their input argument. So, we need to have a way to tell these programs exactly which file or directory we would like them to perform some operations on. There are two methods for addressing a file or directory:
  - **Full path addressing:** In this method, all path elements are written from the root directory with each two consecutive elements separated by a '`/`' character. For example, suppose we have a directory named '`home`' in the root directory and inside '`home`', there is another directory named '`acm`' which itself contains a directory or file named '`uva`'. Using full-path addressing, we should write '[`root`]`/home/acm/uva`' in order to address '`uva`'. For the sake of simplicity, it is decided that '[`root`]' should be omitted from the above phrase thus resulting in '`/home/acm/uva`' for full-path addressing '`uva`'.
  - **Relative Addressing:** Suppose that in the above example, '`acm`' has two directories named '`uva`' and '`avu`' and the user has set the current directory to '`avu`'. Now, if he/she wants to address '`uva`', in addition to full-path addressing, he has a simpler way which is by using the string '`../uva`'. In other words, a directory name of '..' indicates the parent directory of the current directory or the parent directory of the directory which is listed in the address string exactly before its occurrence. So, in the example above, '`../..`' is equal to '`/home`' and '`../../../home/..`' is equal to the root directory. Theres also another directory name '`.`' which means the current directory. So, again in the example above, assuming that the current directory is '`/home/acm,`' '`./uva`' is equal to '`/home/acm/uva`'. Note that the user have the choice to omit '`./`' from this kind of addresses. So, '`uva`' is exactly equal to '`./uva`'.

Now that we know enough about the file system and addressing, we can see what the commands are.

Commands

Usage	Description
`cd` *path*	`cd` is the simplest command in our system. `cd` changes the current directory to *path* which is a directory addressed using either full-path or relative addressing methods. If *path* does not exist, the error message "*path* `not found`" should be printed in the standard output.
`touch` *filename* [`-`*size*] [`-h`]	`touch` command is used to create or modify a file. *filename* is actually a path with its last element equal to the name of the file to be created. All the elements of the path except the last element should form a valid address to an existing directory and the last element should be a valid filename. If the indicated filename does not exist in the indicated path, a new file is created with its size equal to *size* argument of the command. If the argument is not given then the size of the newly created file is assumed to be equal to zero. Also `-h` argument tells the bash to set the `hidden` attribute for the newly created file. In the case that the given filename already exists in the indicated path, the file should be deleted and recreated with the new arguments. In the case that the indicated path of the file to be created does not exist, touch command should print the error message "*path* `not found`" on the standard output. And in the case that there exists a directory in the indicated path with the same name as the file to be created, the error message "`a directory with the same name exists`" should be printed on the standard output.
`mkdir` *path* [`-h`]	Creates a directory with the address equal to *path* argument. The last element of the address is the name for the new directory. The other elements of the address should form a valid address and should address an existing directory. In the case that these elements do not meet these conditions, the error message "*path* `not found`" should be printed in the standard output. Also, if a file or directory exists in the given path with the same name as the directory to be created, the error message "`file or directory with the same name exists`" should be printed on the standard output. Note that if `-h` is specified as an argument, then the newly created directory is given the `hidden` attribute.
`find` *filename* [`-r`] [`-h`]	Searches for files or directories. *filename* should be a valid address with the last element equal to the name of the file or directory to be searched for. The other elements of the address should address an existing directory which is the directory to be searched or else the error message "*path* `not found`" is printed on the standard output. If `-r` is specified then all of the directories inside the given address should also be searched recursively. By default, `find` does not search within the files or directories having the `hidden` attribute unless `-h` is specified as an argument. For each file or directory found with the specified name, exactly one line is printed in the standard output containing the full-path of the file or directory, its size, the word `hidden` if the `hidden` attribute is set for the file and the word `dir` if the directory attribute is set for the file. All these properties should be separated by single space characters. The results should be sorted by lexicographical order of full-path addresses which means that in order to compare two files, the path elements of their full-path addresses are lexicographically compared to each other from left to right. In the case no such file or directory is found, the error message "`file not found`" should be printed on the standard output.
`ls` [*path*] [`-h`] [`-r`] [`-s`] [`-S`] [`-f`] [`-d`]	`ls` is a very popular command used to list the contents of [part of] the file system. In its simplest form, `ls` does not take any arguments. In this case, all of the files and directories in the current directory that do not have the `hidden` attribute are listed in a similar format to that of `find` command. In the case there is no such file or directory in the current directory, it prints the message "`[empty]`". If *path* is specified as an argument and is the address of an existing directory, then `ls` performs its operation on *path* instead of the current directory. Among the other arguments, if `-h` is specified, then the files and directories with `hidden` attribute are also shown. In the case `-r` is specified, `ls` performs its operation on the directories within the original directory to be searched recursively. If `-s` is specified, the results are first sorted in non-decreasing order by size. There, in the case there are two files having the same sizes, the two files should be ordered in a manner similar to `find` command. As an argument, the functionality of `-S` is similar to `-s` instead of the fact that `-S` sorts the files by size in non-increasing order. Argument `-d` forces `ls` to only show the directories in the result while `-f` forces `ls` to only show the files that do not have the directory attribute.
`pwd`	Prints the full-path address of the current directory on the standard output.
`exit`	Terminates the session and exits BASH
`grep` "*string*"	`grep` is a special command with a special usage form. In detail, it takes an string and searches its standard input for all occurrences of that string and outputs all the lines of its standard input that contain those occurrences on the standard output. In this special version of `grep` in our BASH, the standard input can only be fed to it by the means of a technology named piping. Piping is a technology that can be used to attach the standard output of one command to the standard input of another one. In more details, all the outputs generated by the first command are passed to the second command as standard input. Pipes can be used in the form *command*1 `|` *command*2. Executing this phrase causes BASH to execute command1 first and then pass its standard output to command2 and execute command2 respectively. So, to feed `grep` with some input, we should use pipes as stated above. For example, "`ls	grep "acm"`" searches the result of `ls` command for any lines containing the word '`acm`'.

At last note that the mandatory arguments of all the commands should be passed with the specified order but the optional arguments can be passed in any order. Any of the above commands if not used according to the usage form specified, produce the error message "`bad usage`" on the standard output e.g. if a command is passed extra or redundant arguments. Also, if a command is entered which is not among the above commands, the error message "`no such command`" should be printed on the standard output. Note that these two error messages cannot be piped to `grep`.

Now that you exactly know how our BASH works, in order to warm-up yourself for implementing the real BASH, just write a program that simulates a chain of commands and produces outputs. It is guaranteed that none of the commands have two (or more) type of errors simultaneously.

You are advised to read the sample input/output carefully to see how these commands work.

Input

Input consists of several work sessions. Each session consists of several lines, each containing a command to be executed. Length of a line containing a command is no more than 2048 characters. Also, there may be arbitrary number of space characters before and after the command and between its arguments. Each session is terminated with an "`exit`" command. Input is terminated by end-of-file. You must assume that in the beginning of each session, the default directory is the root directory and the file system is empty.

Output

For each session, your program should produce the output of each command appearing on the input.

Sample Input

```
pwd
cd acm
mkdir ./acm
ls ./acm
cd acm
pwd
touch acm -h -1000
cd ..
cd /
grep
ls -r -s
ls -r -h
find acm -h -r | grep "1000"
exit
```

Sample Output

```
/
path not found
[empty]
/acm
bad usage
/acm 0 dir
/acm 0 dir
/acm/acm 1000 hidden
/acm/acm 1000 hidden
```