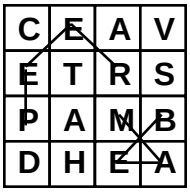
Seventh Annual
University of Oregon Programming Competition

Saturday, May 3, 2003

Common Boggle

Boggle is a word game by Parker Brothers in which you form words from adjacent letters in a 4×4 square grid of letters. Letters are adjacent if they are next to each other horizontally, vertically, or diagonally. The letters are on the faces of sixteen letter cubes, and the cubes are shaken up and then placed in a tray to start a new game. A word cannot use a cube more than once. However, the same letter may appear more than once in a word if the letter appears on multiple cubes in the formation of the word. The cubes used in Boggle result in a reasonable distribution of vowels and consonants.



Example words for this puzzle: PEER BEAM

In this problem, you are given *two* Boggle game configurations and you must find all of the *words in common* between the two Boggle games. For these purposes, words are any sequence of letters from adjacent cubes – the sequence does not have to be a legal word in any dictionary. For each pair of games, you will only need to find the common words of a particular length, which could range from two to five.

The first line of input to this problem will contain an integer n indicating the number of Boggle game pairs to follow Each pair of games will begin with a line with a single integer $\ell \leq 5$, the length of the common words you are to find. The game configurations will follow and consist of the letters in each row of each game, with each pair of rows on a separate line.

Output should be a list, in alphabetical order, of the common words. Repeated words should not appear in the list, even if they can be formed in different ways in each of the games. If there are no words in common, your program should print

```
There are no words in common.
```

Sample Input

```
2
4
LKOG ETPM
SAOV NOGF
BMRO HEAG
RKFI RLDI
3
EQRT UNLE
TZTO DEFI
TLLR EPEM
CPOB ZGNT
```

Sample Output

```
GOAL
LAOG

There are no words in common.
```

Lifting Transformations

A common task in reverse engineering of software is the construction of *architectural transformations*—manipulations of the design-level structure of a system, based on properties of lower-level facts. Among these transformations is *lifting*:

> *Given a description of some low-level structural relationship in a software system, produce a corresponding relationship between high-level components.*

We are specifically interested in lifting the *call graph* (which indicates when a function calls another function) to higher files and directories. Thus suppose that file `X.foo` contains function `f` and file `Y.foo` contains function `g`. Then if `f` calls `g` we can lift that relation to the relations (viewed as ordered pairs): (`X.foo,g`), (`f,Y.foo`), (`X.foo,Y.foo`). If, furthermore directories `UU` and `VV` contain, respectively, `X.foo` and `Y.foo`, then lifting would also imply any of the relations (`UU,g`), (`f,VV`), (`UU,Y.foo`), (`X.foo,VV`), (`UU,VV`). If `UU` or `VV` are contained in higher-level directories, similar relations are propogated.

Your job here is to write a program that produces the `tlcall`, the relation that the call graph ultimately induces on the *top-level entities* (a subset of the files and directories that a user has selected for viewing).

Notation. Here are some terms that your program might encounter:

- "`toplevel AA t`" signifies that `AA` is one of the top-level entities (the extra "`t`" is included for techical reasons not relevant to your program).
- "`funcdef foo.c f`" signifies that function `f` is defined in file `foo.c`,
- "`sourcecall f g`" signifies that function `f` calls function `g`,
- "`contain A foo.c`" signifies that directory `A` contains file `foo.c` (files can't contain files!),
- "`contain AA A`" signifies that directory `AA` contains entity `A` (an *entity* is either a file or directory).

The first line of input to this problem will contain an integer n indicating the number of systems to follow. Each system instance will start with a line containing an integer m that indicates how many facts will be presented about the system. Each of the succeeding m lines will contain a triple respresenting one such fact. The triples are of the form

$$relation \quad source \quad target$$

(separated by 1 to 5 white spaces).

For each system, the output should be the relation `tlcall`, with one triple per line Base your answer on the relations `funcdef`, `sourcecall`, `contain`, and `toplevel`. If no `tlcall` can be inferred, output the line:

```
** No information at this level **
```

There should be a blank line between systems.

Sample Input

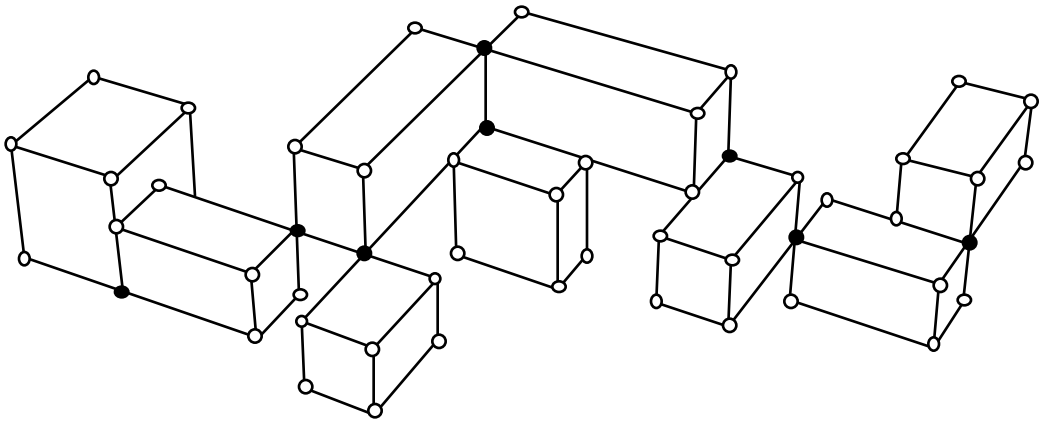
```
2
11
funcdef    foo.c   f
funcdef    foo.c   g
funcdef    bar.c   h
sourcecall f   g
sourcecall f   h
sourcecall h   g
contain    AA   A
contain    A foo.c
contain    CC bar.c
toplevel   AA t
toplevel   CC t
7
funcdef    bar.c   h
funcdef    foo.c   f
sourcecall f   h
contain    A  bar.c
contain    BB foo.c
toplevel   DD t
toplevel   BB t
```

Sample Output

```
tlcall   AA CC
tlcall   CC AA
tlcall   AA AA

** No information at this level **
```

Navigating BoxSpace



BoxSpace is a city constructed entirely of three-dimensional, box-shaped structures. The boxes may be of various sizes, and they may be adjacent to each other, but boxes may never overlap.

Because the city has grown so large, residents now use air-powered vehicles to travel from point to point. These vehicles can only be driven along tracks that run along box edges. Each track may be driven in either direction. Vehicles may only transfer to another box at one of eight stations, one at each corner of a box *but only when two boxes share the same station*. In addition, all travel starts and ends at a station.

Your company is working on a top-secret computer program for navigation within BoxSpace. The program will store a map of BoxSpace and, given a source (taken from GPS coordinates) and destination (given by voice command), drive a vehicle along the shortest route to the destination.

Your task is to build a prototype of this navigational program.

The first line of input to your navigational program is an integer, n, indicating the number of BoxSpace instances to follow. Each instance consists of the following parts

i. A map of BoxSpace. The first line of the map contains the number, b, of boxes. Each of the following b lines contain the description of one box. A box is described by six numbers. The first three are, respectively, the x-y-z coordinates of the station in the lower left corner of the box (minimum x, y, and z coordinates). The second three are the width, height, and depth of the box.

ii. On a single line, six numbers denoting the coordinates of a start station followed by the coordinates of the destination station.

For each BoxSpace instance, your output should be a list of (parenthesized) coordinates that represents the shortest path from the beginning station to the end. The first set of coordinates should represent the start station and the last set coordinate should be the destination. Between these two stations you should print the coordinates of any other stations visited. Along with each coordinate you should print a running total of the distance to be traveled. You should leave a blank line between instances.

Sample Input

```
2
3
0 0 0 5 4 2
5 0 0 5 10 5
0 4 0 3 2 1
0 0 0 3 6 1
5
0 0 0 5 4 2
5 0 0 5 10 5
0 4 0 3 2 1
10 0 0 10 5 3
10 5 0 2 3 10
3 6 1 12 5 10
```

Sample Output

```
(0,0,0) 0
(0,4,0) 4
(3,4,0) 7
(3,6,0) 9
(3,6,1) 10

(3,6,1) 0
(3,6,0) 1
(3,4,0) 3
(0,4,0) 6
(0,0,0) 10
(5,0,0) 15
(10,0,0) 20
(10,5,0) 25
(12,5,0) 27
(12,5,10) 37
```

Cubic Conclusions

In this problem you are given A^3 for some very large integer A and you need to determine A. Since that seems such an exhausting task, you are asked only to find the last m digits ($m \leq 20$) of A, in which case you only need to be told the last m digits of A^3.

The first line of input to this problem will consist of an integer n, indicating the number of problem instances to follow. Each instance will consist of a sequence of m digits for some $m \leq 20$, the last digit will always be 1 and first digit will never be 0. (However, that does not preclude 0 as the first digit of the output.)

Your output for each instance should be a line containing the last m digits of an integer whose cube ends with the last m digits of the input.

Sample Input	Sample Output
4	1
1	41
21	901
701	0011
1331	

Flagging Teams

The popular camp game, Capture-the-Flag, can be played with two teams of any size and so the whole camp participates. However, the counselors often find it difficult to form the two teams since, by late Summer, there are pairs of campers who cannot stand each other. Such pairs cannot be assigned to the same team since Capture-the Flag demands close teamwork.

In this problem, you are to design a program that will take as input $n \leq 100$ campers and assign them to two teams with $n/2$ campers in each (n will always be even) while making sure not to assign an incompatible pair of campers to the same team.

The first line of input to this problem will consist of an integer c, indicating the number of camps to follow. Each camp instance will start with a pair integers n, i where n is number of campers and i is the number of pairs of campers that do not get along. To save your program the trouble of reading the long list of campers' names, the campers will have been assigned numbers 1 through n. Hence, the next i lines will each contain a pair of integers corresponding to incompatible campers.

For each camp instance, you are to list two possible teams for Capture-the-Flag, with one team per line and each team in numerical order. Instances must be separated by a blank line. If it is impossible to form teams for the camp, you should output

```
We cannot play Capture-the-Flag.
```

Sample Input

```
3
4 2
1 2
1 3
6 4
1 2
1 5
3 4
3 6
8 5
1 2
1 3
1 4
1 5
1 6
```

Sample Output

```
1 4
2 3

1 4 6
2 3 5

We cannot play Capture-the-Flag.
```