University Interscholastic League

Computer Science Competition

2003 Regional Programming Set

I. General Notes

1. Do the problems in any order you like. They do not have to be done in order from 1 to 10.
2. All problems have a value of 6 points.
3. Some problems have a reference to columns of an input line, such as column 1 or columns 1-3. In these cases column is referring to the character position on the input line. Column 1 refers to the first character position on the line, while columns 1-3 refer to the first three positions on the line.
4. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.
5. Your program should not print extraneous output. Follow the form exactly as given in the problem.

II. Point Values and Names of Problems

Number	**Name**	**Point Value**
Problem 1	Make 'em Look Good	6
Problem 2	Master Index	6
Problem 3	The Path Less Traversed	6
Problem 4	Bomberman!	6
Problem 5	Does This Equate?	6
Problem 6	The Logging Industry	6
Problem 7	Desperately Seeking Austin	6
Problem 8	Every Rose Has Its Thorn	6
Problem 9	There is No 'I' in TEAM	6
Problem 10	Freddy's Fast Fingers	6
Total		**60**

Problem 1

Make 'em Look Good

6 Points

Program Name: baseball.cpp   Input File: baseball.dat

Introduction

You are the Jumbotron operator for ~~Enron Field~~ ~~Astros Field~~ Minute Maid Park. You have been told that when the Houston Astros are batting, you want to display their batting statistics in the best possible light. To do this you should display on the Jumbotron the batter's batting average X for the last Y games, where Y is the game in the last ten games for which X would be the highest. Batting averages are calculated by dividing the total number of batting attempts by the total number of hits and rounding to the nearest thousandths. For example, to calculate a batter's batting average for the last 3 games, you would divide the total number of batting attempts for the last 3 games by the total number of hits for the last 3 games, then round to the nearest thousandth.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 3 components:

1. *Start line* - A single line, "START"
2. The next ten lines will represent the batter's performance for the last ten games. The first line will refer to the batter's last game, the second line to the batter's second-to-last game, and so on. Each line will be in the format, "A B", where:
   A : (0 <= A <= 9) is the number of hits the batter had that game.
   B : (3 <= B <= 9) is the number of batting attempts the batter had that game
   and, obviously, B >= A.
3. *End line* - A single line, "END"

Output

For each data set, there will be exactly one line of output. The output will be a single line in the format "BATTING X FOR THE LAST Y GAME(S)", where X and Y are calculated as described. If there are multiple values of Y that would result in the highest X, choose the value for Y that is the greatest. For example, if a batter were batting .500 for the last 3 games, 4 games, and 6 games, the value for Y would be 6. Display batting averages to three decimal places and do not display any leading zeroes, but display any leading ones.

Example: Input File

```
START
1 3
1 3
3 4
2 4
1 3
3 5
1 7
4 5
1 4
2 5
END
START
4 4
2 3
3 5
1 3
1 4
1 3
2 4
4 5
3 6
3 3
END
```

Output to screen

```
BATTING .500 FOR THE LAST 6 GAME(S)
BATTING 1.000 FOR THE LAST 1 GAME(S)
```

Problem 2

Master Index

6 Points

Program Name: index.cpp **Input File: index.dat**

Introduction

You are a student of Master Index, a world-renowned author of indices. Master Index is retiring soon and will recommend you as his replacement, if you are able to prove yourself. The rules of indexing are simple:

Given an index term and a list of page numbers, an index entry consists of:

1. The index term, followed by a single comma.
2. A list of sequentially-ordered page numbers the term appears on. Each page number will be separated by a single comma, unless there is a series of two or more consecutive page numbers, in which case instead of each page being listed, the first page number in the series is listed, followed by a single dash and the last page number in the series.

To prove yourself to Master Index, you wish to write a program that will write index entries.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 3 components:

1. *Start line* - A single line, "START A B", where A is the index term and B (1 <= B <= 100) is the number of pages on which the term appears. An index term will consist of a single word containing one to twenty alphanumeric characters.
2. Each of the next B lines will contain a page number the index term appears on. The page numbers will be sequentially ordered.
3. *End line* - A single line, "END"

Output

For each data set, there will be exactly one line of output. The output is the index entry, formatted according to the description in the introduction.

Example: Input File

```
START broccoli 7
138
140
141
142
144
145
150
END
START cheese 1
138
END
START doughnuts 2
28
30
END
```

Output to screen

```
broccoli,138,140-142,144-145,150
cheese,138
doughnuts,28,30
```

Problem 3

The Path Less Traversed

6 Points

Program Name: path.cpp Input File: path.dat

Introduction

The following is a brief recap of Unix pathnames:

- UNIX pathnames use the forward slash (/) as a directory separator.
- The root directory has a pathname of "/".
- "." refers to "this directory." It is generic across the directory structure, so whatever directory you're in can be referred to as ".". "." is pronounced "dot."
- Similarly, ".." refers to the directory immediately above the current directory. This directory is also called the parent directory. Again, this reference is generic across the file system, so ".." refers to the parent directory of your current directory, wherever that may be. ".." is pronounced "dot-dot."

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 1 component:

1. *Start line* - A single line, "A B", where:

A will be a path name representing a directory tree. A will start with a '/' (referred to as the "root" directory), followed by directory names delimited by a single '/'. A directory is considered a subdirectory of the directory immediately preceding it in the list (conversely, the directory preceding a directory in the list is considered its parent directory). For example, if A is "/subdir1/subdir2", the directories are "/", "subdir1", and "subdir2", where "subdir1" is a subdirectory of "/", and "subdir2" is a subdirectory of "subdir1", "subdir1" is the parent directory of "subdir2", and "/" is the parent directory of "subdir1". Directory names will consist of alphanumeric characters only. The maximum length of A will be 100 characters.

B will be an absolute path name to be traversed. B will always start with a '/' followed by a series of '.', '..', and directory names, delimited by a single '/'. For example, B could be "/./subdir1/../subdir1". The maximum length of B will be 100 characters.

Output

For each data set, there will be exactly one line of output. If B is a valid path name, the output will consist of the absolute path name of the directory reached after traversing B for the directory tree represented by A. In the example above, the output would be "/subdir1". Traversal is accomplished reading B left to right as follows:

A starting '/' in B (which is always the case) means you start in the root directory.
A '.' in B means you traverse to the directory you are currently in.
A '..' in B means you traverse to the parent directory of the directory you are currently in, unless you are in the root directory, in which case you traverse to the root directory.
A directory name in B means you traverse to that directory, if it is a subdirectory of the directory you are currently in. Otherwise, B is not a valid path name.

If B is not a valid path name, the output will consist of a single line with the statement "INVALID DIRECTORY".

Example: Input File

```
/subdir1/subdir2 /./subdir1/../subdir1
/subdir1/subdir2/subdir3 /./././subdir1/subdir2/subdir3/../../subdir2
/subdir1/subdir2/subdir3 /./././subdir1/subdir2/subdir3/../../subdir3
```

Output to screen

```
/subdir1
/subdir1/subdir2
INVALID DIRECTORY
```

Problem 4 | **6 Points**

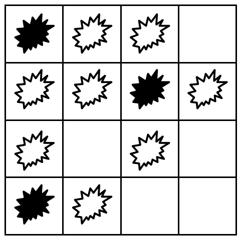
Bomberman!

Program Name: bomberman.cpp   Input File: bomberman.dat

Introduction

You are a huge Bomberman fan, but the only problem is that you aren't necessarily that good at it. You are particularly bad at the levels where you have to avoid raining bombs from the heavens. You have played each level so many times that you know exactly where the bombs are going to drop, but your only problem is that it seems no matter what sequence of moves you perform, you always seem to get clobbered! You have come up with the idea that if you could figure out how to find a spot that is never bombed, then you could go there and not have to move at all and pass the level! Luckily your programming skills are much better than your Bomberman skills, so you quickly whip up a program to calculate the safe spots given the bomb trajectories.

The area map that Bomberman is in can be broken up into a grid of squares. Bomberman can occupy one square and the trajectory of a bomb is the series of squares in which the bomb affects. This includes the square in which the bomb lands, along with one square in each adjacent direction (up, down, left, and right). Bombs do not affect diagonal squares, and can only affect squares that are located on the area map. If Bomberman occupies one of these affected squares, he is toast! Here is an example of a 4x4 area map where the coordinates of each square go from 0-3 (starting from the top left-hand side of the area map), and the locations of the resulting trajectories of three bombs given they were dropped on squares (0,0), (1,2), (3,0):



The safe spots for Bomberman would then be squares (0,3), (2,1), (2, 3), (3,2), and (3,3). In other words, if Bomberman would stay in any of these spots, he could avoid the barrage of bombs without moving, hooray!

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 3 components:

1. *Start line* - A single line, "START X Y Z", where X is number of rows in the area map, Y is the number of columns in the area map, and Z is the number of bombs that will be dropped; where (1 <= X <= 10), (1 <= Y <= 10), and (1 <= Z <= 5).

2. *Bomb Positions* - A single line consisting of Z positions on the area map that a bomb will be dropped. Each bomb position will be of the format (Bx,By), where 'Bx' is the row coordinate and 'By' is the column coordinate that the bomb will be dropped in. If there is more than one bomb position, each bomb position will be separated by a space.

3. *End line* - A single line, "END"

Note: *All bomb position coordinates start at 0.*

Output
For each data set, there will be exactly one line of output. Each line of output will consist of a list of coordinates that are not affected by any of the dropped bombs, and are thus safe spots for the Bomberman to stand. Each safe position will be printed out in the format (Sx,Sy), where 'Sx' is the row coordinate and 'Sy' is the column coordinate of the safe position. The safe coordinates shall be printed in ascending order starting with Sx, then followed by Sy. If there is more than one safe position for a given data set, they will be separated by a single space. If no safe positions exist, the string "BOMBERMAN'S TOAST!" will be printed instead.

Note: *All safe position coordinates start at 0.*

Example: Input File
```
START 4 4 3
(0,0) (1,2) (3,0)
END
START 5 3 5
(0,0) (0,2) (2,1) (4,0) (4,2)
END
```
Output to screen
```
(0,3) (2,1) (2, 3) (3,2) (3,3)
BOMBERMAN'S TOAST!
```

Problem 5

Does This Equate?

6 Points

Program Name: equate.cpp   Input File: equate.dat

Introduction

You are an algebra tutor. One of your most tedious duties is checking your students' solutions to basic algebraic equations (you're a lazy tutor), so you will write a program that takes in a basic algebraic equation and gives back the solution (solves for the variable 'x'). This should make your job easy!

Your program should be able to solve any basic algebraic equation involving one variable, where a basic algebraic equation is defined as the following:

x = <*expression*>	Where <*expression*> is an arithmetic expression involving one or more arithmetic operations including addition, subtraction, and/or multiplication. Note that all arithmetic operations operate on integers, all operations will appear consecutively with spaces between them, each operand and operator will be separated by a space as well, each <*expression*> will consist of at most 10 arithmetic operations, and that the normal rules of operator precedence apply.

Notes: All operands of a basic algebraic equation will be a postive integer within the inclusive range of [0, 10000]. All intermediate and final results will be in the inclusive range of [-10000, 10000].

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 1 component, consisting of the actual basic algebraic equation as specified above.

Output

For each data set, there will be exactly one line of output. The output shall be formatted as follows:

"x = <result>"   (where 'result' is the solution to the basic algebraic equation, the value of 'x')

Example: Input File

```
x = 100
x = 200 + 4 - 8
x = 5 * 10 - 2 + 10 * 3
x = 5 - 10
```

Output to screen

```
x = 100
x = 196
x = 78
x = -5
```

Problem 6

The Logging Industry

6 Points

Program Name: log.cpp   Input File: log.dat

Introduction

After starting work at a commercial database company, you've been assigned to work on a recovery utility for crashed databases. Your utility will be given a restored version of the last good backup of the database along with the transaction log containing operations that were performed since the last backup. Your utility must "replay" the transaction log to bring the database up-to-date.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets:

1. *Database Start Line* – A single line, "DATABASE *n*", where $1 \le n \le 100$ and *n* is sequential identifier for this data set (starting with 1 for the first data set and incrementing by 1 for each data set thereafter).

2. *Database Entries* – A series of zero to 10 lines (inclusive) representing the most recent restored backup of the database. Each entry will follow the format `Name Value`. The Name is a string composed of 1 to 20 (inclusive) alphabetical characters (no spaces). The value is an integer in the range $-10000 < \text{Value} < 10000$.

3. *Log line* -- A single line, "LOG". [You can assume that none of the Names for any of the *Database Entries* will be "LOG".]

4. *Log Entries* – A series of zero to 10 lines (inclusive) representing the transaction log entries which occurred since the most recent backup of the database. There will be three types of entries:

   Inserts – "Insert Name Value"
   Deletes – "Delete Name"
   Updates – "Update Name Value"

   [Note: The operations will only appear when they make sense. Inserts will only appear if there is no existing record in the database with a matching Name, Deletes will only appear if there is a record with a matching Name to be deleted, and Updates will only appear if there is a record with a matching Name that should have its Value updated.]

Output

Output for each data set will consist of two parts. The first is an exact replica of the *Database Start Line* for the data set. The second will be the sorted contents of the database after replaying the log. Sorting should occur in ascending order according to the ASCII values of the characters in each entry's *Name* field. [Please assume that, within a single data set, the *Name* fields of entries in the database at any time during the transaction replay will be unique.]

Example: Input File

```
DATABASE 1
James 1000
Marc 1000
Tim 1000
LOG
Insert Laura 100
Delete Tim
Update Marc 500
DATABASE 2
Nochange 0
LOG
DATABASE 3
LOG
Insert Allnew 1
```

Output to screen

```
DATABASE 1
James 1000
Laura 100
Marc 500
DATABASE 2
Nochange 0
DATABASE 3
Allnew 1
```

Problem 7

Desperately Seeking Austin

6 Points

Program Name: findut.cpp   Input File: findut.dat

Introduction

In a recent prophetic dream, your team earned a trip to the UIL finals. However, the penny-pinching principal at your school won't authorize the trip to Austin until you can prove that your team will be responsible and choose the most economical route to UT Austin.
In response, your team is writing software that will scan Texas highway maps and determine the shortest route. The other members of the team are writing the OCR software, and you are authoring the module that will process city connection and road length data to determine the shortest path between any two specified towns, which should also be the path that uses the least amount of valuable school-funded gasoline.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets:

1. *Dataset Start Line* – A single line, "DATASET n", where $1 \le n \le 10$ and n is the number of lines in this data set (not including the *Dataset Start Line*).

2. *Town List* – A single line containing a number ($m \mid 2 \le m \le 10$) followed by a space-separated list of m town names. The first name will be your starting town, and the second name will be your destination. All other towns are possible stops along your route. Each name will be from one to twenty alphabetical characters in length.

3. *Road List* – A series of (n - 1) lines representing the lengths of known roads between pairs of towns from the *Town List*. Each line will list two town names followed by the length of a known road ($d \mid 1 \le d \le 1000$) connecting them. A given pair of towns will appear in this list at most once and possibly never.

Output

Output for each data set will consist of two parts. The first is an exact replica of the *Dataset Start Line*. The second will be the list of towns visited followed by the total distance traversed. The towns should be listed in the order they would be visited when traveling from the starting town to the destination by the shortest possible path. You may assume that there will be one and only one shortest path.

Example: Input File

```
DATASET 5
4 Dallas Austin Houston SanAntonio
Dallas SanAntonio 275
Dallas Houston 238
SanAntonio Austin 79
Houston Austin 162
DATASET 2
2 MyHouse YourHouse
MyHouse YourHouse 1
DATASET 4
3 MyHouse YourHouse Somewhere
MyHouse YourHouse 300
YourHouse Somewhere 100
MyHouse Somewhere 100
```

Output to screen

```
DATASET 5
Dallas SanAntonio Austin 354
DATASET 2
MyHouse YourHouse 1
DATASET 4
MyHouse Somewhere YourHouse 200
```

Problem 8

Every Rose Has Its Thorn

6 Points

Program Name: rose.cpp   Input File: rose.dat

Introduction

"...the roses fearfully on thorns did stand..." - William Shakespeare, playwright, 1564–1616
"...every rose has its thorn..." - Poison, 80s rock band

Oh, how you wish to win the heart of your true love with a single red rose, but you don't want to prick your finger when plucking the flower from the rosebush. Given the location of thorns on a 10cm rose stem, you must determine if you can pick the rose without touching a thorn. You are able to do this if there exists a 1.25cm stretch of no thorns at the same heighth on each of the four sides of the rose stem.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 3 components:

1. *Start line* - A single line, "START X", where:
   X : (1 <= X <= 40) is the number of thorns on each side of a four-sided rose stem.
2. Each of the next four lines will contain a list of X thorn locations for a side of the rose stem. Thorn locations will be represented by a decimal number (to 2 decimal places) that signifies the distance in centimeters from the bottom of the stem to the thorn, and the list of these locations will be delimited by a single space. Locations will be between 0.00 and 10.00, inclusive.
3. *End line* - A single line, "END"

Output

For each data set, there will be exactly one line of output. If you able to pick the rose without touching a thorn, the output will be a single line with the statement, "A ROSE FOR MY LOVE". Otherwise, the output will be a single line with the statement, "A THORN FOR MY TROUBLES".

Example: Input File

```
START 4
1.00 5.00 6.00 8.00
2.00 3.00 4.00 7.00
0.25 0.50 5.75 8.75
0.50 0.75 4.00 4.25
END
START 5
2.00 3.00 4.00 5.00 6.00
5.00 6.00 7.00 8.00 9.00
2.25 3.43 6.89 7.00 8.40
1.24 4.58 6.78 7.12 8.34
END
```

Output to screen

```
A ROSE FOR MY LOVE
A THORN FOR MY TROUBLES
```

Problem 9

There is No 'I' in TEAM

6 Points

Program Name: set.cpp **Input File: set.dat**

Introduction

As manager of a high-profile project for Give Us Venture Capital Inc. (motto: "We code the stuff that makes you money"), you are constantly under pressure from your supervisor to meet programming deadlines. You notice that some of the programmers on your team work very well alone, but other programmers work extremely well with particular members on the team. This synergy allows them to code better and faster than some of the other programmers working alone. You need to pick the three programmers that will give you the maximum amount of work units done in order to meet the next looming deadline.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 3 components:

1. *Start line* - A single line, "START X", where X is the number of programmers to choose from (3 <= X <= 9).
2. *Programmer list* - A series of X lines. Each line lists, for a single programmer, a list of work unit integer values, delimited by a single space. The first value indicates a number of work units the programmer accomplishes if programmer 1 is on his team, the second value indicates a number of work units the programmer accomplishes if programmer 2 is on his team, and so on, up to X values.
   Each of these values, Y, will be: -500 <= Y <= 500.
3. *End line* - A single line, "END"

Output

For each data set, there will be exactly one line of output. The output will be a single line in the format "A B C D", where A, B, and C (in order from least to greatest) are the numbers for the three programmers that would accomplish the greatest number of work units if they worked together on a team. D, the number of work units a given team accomplishes, is calculated by adding the total number of work units each programmer on the team accomplishes. The number of work units each programmer on the team accomplishes is calculated by totalling the number of work units he accomplishes if each of the programmers (including himself) were on the team. There will be no "ties" (two different combinations of programmers that would result in the greatest number of work units).

Example: Input File

```
START 4
500 0 0 0
0 200 200 200
0 200 200 200
0 200 200 200
END
START 5
500 -500 -500 -500 500
-100 300 100 100 500
-100 100 300 100 500
-100 100 50 300 500
0 0 0 0 0
END
```

Output to screen

```
2 3 4 1800
2 3 5 1800
```

Problem 10 **6 Points**

Freddy's Fast Fingers

Program Name: typing.cpp   Input File: typing.dat

Introduction

Freddy likes to type. And he types fast. Sometimes he'd like to know how long it is going to take to type something before he starts. He observes his typing habits:
Given the standard QWERTY keyboard layout (Freddy has yet to discover the advantages of the Dvorak layout):

```
    Column
    0 1 2 3 4 5 6 7 8 9
R 0|q w e r t y u i o p
o 1|a s d f g h j k l ;
w 2|z x c v b n m , . /
  3|-----Space bar-----
```

Freddy uses his fingers accordingly (in typical typing fashion):

Finger	**Types**	**Starts on**
Left little finger	qaz	a
Left ring finger	wsx	s
Left middle finger	edc	d
Left index finger	rtgfvb	f
Right index finger	yuhjnm	j
Right middle finger	ik,	k
Right ring finger	ol.	l
Right little finger	p;/	;

(Not shown in the table is Freddy's right thumb which starts on the space bar and types the space character)

Freddy's fingers start on the keys according to the table. However, the instant he starts typing, each of his fingers moves to the next character that finger has to type. This movement occurs simultaneously. When one of his fingers is on the next character to be typed, it pushes that key and that finger immediately proceeds to the next character that finger needs to type, simultaneously with any other fingers that may be moving to their next character to type, or pushing keys. This continues until Freddy has finished typing all the characters.

The time it takes for a finger to move from one key to another can be calculated (in ms) as follows:
((| destination column - starting column |)) * 50) + ((| destination row - starting row |) * 50)
using the rows and columns from the keyboard layout designated above.

The time it takes to press a key is 10ms.

Input

Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be **no blank lines** separating data sets.

A single data set has 1 component:
Typing Copy - A single line of 1-200 characters from the QWERTY keyboard layout designated in the introduction.

Output

For each data set, there will be exactly one line of output. The output will be the integer "A", where "A" is the amount of time in ms it will take for Freddy to type the characters inputted.

Example: Input File

```
asdf jkl;
the quick brown fox jumps over the lazy dog.
```

Output to screen

```
90
1280
```