

Problem 1: Boating

A classic puzzle has a farmer, a wolf, a goat, and a cabbage try to cross a river with a boat which must be operated by the farmer and that can only hold one other passenger. What makes this problem interesting is that the wolf and goat or the goat and cabbage cannot be left alone together on either shore. The problem is to plan a series of trips across the river that will eventually get everyone to the other side.

You are to write a program that will solve a generalization of this problem. Several passengers approach a river at which there is a boat that can carry a specified number of passengers. Each passenger has a name and may be specified as a peacemaker. Optionally a passenger may provide a list of names of other passengers whom they do not want to be present with unless a peacemaker is present also. Due to an epidemic of seasickness this restriction only applies on shore; everyone is too busy with seasickness on the boat to worry about it (i.e., the goat would not want to be with the wolf on shore, but the wolf is too sick to bother the goat on the boat).

The first line of input to your program will consist of three integers; the first specifies the number of passengers that can be in the boat at the same time, the second specifies the number of passengers that wish to cross the river, and the third number specifies the maximum number of trips the boat can take. For each passenger, there is a name (a single word consisting of alphanumeric characters), at least one blank, the letter T or F indicating whether the passenger can operate the boat, at least one blank, the letter T or F indicating whether the passenger is a “peacemaker”, and optionally at least one blank and a list of names of other passengers whom this passenger doesn’t want to be present with unless a peacemaker is present also.

Your program should print out the minimum number of trips that the boat must make to get all passengers to the other side. If there is no way to get all the passengers safely to the other shore using the specified number of trips or less, your program must print the message “Impossible”.

Sample input (the classic puzzle):

```
2 4 10
Farmer T T
Wolf F F
Goat F F Wolf
Cabbage F F Goat
```

Sample output:

```
7
```

Problem Number 2: Squared Digits

Prof. Andrew C. McDigit wrote a program to find all integers between 1 and 1000 inclusive that are the sums of the squares of their digits. For example, $1 = 1^2$ so it is one of the integers Prof. McDigit wanted to find. Prof. To his intense disappointment he discovered that 1 is the *only* such integer.

Prof. McDigit would like to have his program rewritten so that it is possible to specify the amount by which the sum of the squares of the digits may differ from the original number. For example, if Prof. McDigit chose the value 10, 56 now qualifies. Since:

$$5^2 + 6^2 = 25 + 36 = 61$$

and 56 and 61 differ by no more than 10.

Prof. McDigit wants you to rewrite his program. The program will accept a single number as input that specifies how much the number and the sum of the squares of it's digits may at most differ. It will then print all the numbers that satisfy the squared digit property, one number per line in ascending order.

Sample input:

0

Sample output:

1

Sample input:

2

Sample output:

1

2

35

75

Problem Number 3: Make Change

The manager at “Computer R Us” is having a problem with the cashiers working in the store. Although everyone should know how to make change, the cashiers are constantly making mistakes. Since each cashier has a computer at their register, the manager has decided to ask you to write a program that will help the cashiers make change.

Your program will be given a list of the denominations of the coins the cashier has in the cash drawer and a list of amounts. The program will then print the least number of coins that can be used to make up each amount. For example, given an amount like 21 cents, the ways you can make up 21 cents depends on the coins you have (e.g. 2 dimes and a penny, or 4 nickels and a penny, or 3 7-cent coins if such a thing existed).

The first line of input to your program specifies the available denominations of coins as an unsorted list of positive integers separated by white space. The second line contains the amounts for which you have to make change, again as positive integers separated by white space. The program must print the *least* total number of coins selected from the given denominations that can be used to make up each amount on a single line for each amount. If it is impossible to give change for a given amount your program should print a single “Impossible”.

Sample input:

```
7 5 1
21 26
```

Sample output:

```
3
4
```

Sample input:

```
8 7 3 2
26 9 70 1
```

Sample output:

```
4
2
9
Impossible
```

Problem Number 4: Fragment Reassembly

Biologists represent DNA as strings consisting of the letters 'A', 'C', 'G', and 'T'. A *k-length fragment* is a string of length k consisting of (some of) these letters.

You have been asked to write a program that is given a set of k -length fragments and that tries to construct a DNA string from the fragments, all of which have the same length and are different from each other. The DNA string will exactly contain each given fragment and no other k -length fragment.

The input to your program consists of a single line that contains the k -length fragments, separated by at least one blank. The program either prints the DNA string or the message "Impossible".

Sample input:

CCG TAC ACT ACC CGA CTA

Sample output:

ACTACCGA

Sample input:

CCG TAC ACT CGA CTA

Sample output:

Impossible

Problem Number 5: Going Up?

A building planner is in the process of trying to determine the number of elevators to place in a new building. To help the decision making process you have been asked to write a program that simulates the behavior of the elevator.

The elevator will service n floors and is designed to take exactly one time unit to move from one floor to an adjacent floor. Exit and entry on a floor, after the elevator stops, happen instantaneously. The capacity of the elevator is unlimited. Floors in the new building are numbered starting at 1 and end at floor n . On each floor, over time, people arrive and call the elevator. As soon as the elevator arrives at a given floor all the passengers waiting for the elevator on that floor will enter the elevator, regardless of the direction of travel.

The elevator uses what could be called the “Butterfly” algorithm to determine the direction of travel. At the beginning of the day the elevator is stopped at floor 1. It will start to move up when someone calls the elevator to a higher floor, or when someone enters the elevator who wishes to go to a higher floor. As soon as the elevator reaches the top floor, or there is no reason to continue going up (i.e., no one on a higher floor is waiting for the elevator and no one in the elevator wants to go to a higher floor), the elevator reverses direction and repeats the process, this time going to lower floors. If the elevator has no passengers, or no one is calling the elevator, it will wait at the floor it is currently on.

The first line of input to your program will contain two integers; the first specifies the number of floors in the building, and the second the number of people who wish to use the elevator. The remaining lines provide information about each of the people who wish to use the elevator. These lines contain 4 fields; the first is an integer that represents the time at which the passenger makes the request, the second is a string that gives the name of the passenger, the third is an integer that specifies the floor the elevator is being called to, and the fourth is an integer that specifies the floor the passenger wishes to travel to.

The output produced by your program will specify the actions of the passengers. When a passenger enters the elevator you will print the message “XXX enters on floor N ”, where XXX is the name of the passenger and N is the floor they enter the elevator on. If more than one passenger enters the elevator at one stop, the passengers will be processed and listed in order based on the time they called the elevator, and then alphabetically by name. When a passenger leaves the elevator your program will print the message “XXX exits on floor N ”. If more than one person leaves the elevator on a floor, the passengers will be listed alphabetically by name. If there are passengers entering and exiting the elevator on the same floor, the passengers exiting the elevator must be listed first followed by the passengers entering the elevator. The message “Going up” or “Going down” must be printed every time the elevator changes direction.

Sample input:

10 5
0 Paul 1 10
0 PJ 5 10
5 Lisa 9 1
5 Heidi 5 4
10 James 10 9

Sample output:

Going up
Paul enters on floor 1
PJ enters on floor 5
Lisa enters on floor 9
Paul exits on floor 10
PJ exits on floor 10
Going Down
Heidi enters on floor 5
Heidi exits on floor 4
Lisa exits on floor 1
Going up
James enters on floor 10
Going Down
James exits on floor 9

Problem 6: Bitmap Reconstruction

Puzzle magazines call this the Japanese or Chinese puzzle: You have to reconstruct the contents of a rectangular bitmap. You are given the dimensions of the bitmap (a line with two numbers) and then for each row and then for each column the length of each block of contiguous 1-bits (one line per row and one line per column, each containing one or more numbers). Within a row (or a column) there has to be at least one 0-bit between any two contiguous blocks of 1-bits.

Your program should read lines for one problem as described above and should output the bitmap, one line per row that contains periods for 0-bits and capital Bs for 'black' 1-bits.

Sample input:

```
5 5
1 1
1
5
1
1 1
1 3
1
1
1
3 1
```

Sample output:

```
B . . . B
. . . . B
BBBBB
B . . . .
B . . . B
```

Problem 7:

In this problem, you are to create an assembler for the Toy Assembly Language (TAL) that assembles TAL source programs into TAL machine language. Fortunately, all programs that TAL that your assembler will encounter will be syntactically correct. The Toy Machine has two registers, R1 and R2. All integer operations are done in two's complement notation.

TAL instructions are 16-bits long. Bits are numbered from 0 to 15 starting from the right side of the word. The rightmost bit (i.e., bit 0) is the least significant. In the assembled instruction, bits 12-15 will represent the op code; bits 10 and 11 represent registers (binary 01 for register R1, 10 for register R2, and 11 for no register in the instruction); and bits 0-9 represent any address that is necessary for the instruction (in HLT and CIR these should be 0's).

A line of TAL source code will have the following format:

```
LABEL:           OpCodeMnemonic OperandField
```

The label field is optional. A label consists of 1 to 8 uppercase letters. A label cannot have the same spelling as any op-code or pseudo-op mnemonic. If a label is on a line, it must be the first thing on the line and will be followed immediately by a colon (i.e., there is no white space around a label).

An op-code mnemonic may be preceded by white space, and will be followed by at least one white space character if the line contains an operand field. Multiple operands are separated by comma and or white space. In the table below, X represents a symbol or a 10-bit address in 3 hexadecimal digits.

The table below lists each of the op-code mnemonics and the corresponding TAL instruction.

Hex Code	Mnemonic	Meaning
0	HLT	Halt program execution
1	READ X	Read an integer into location X from standard input
2	WRITE X	Write the value in location X to standard output
3	LD X,Rn	Contents of location X are loaded into the Register Rn(n=1 or 2)
4	ST Rn,X	Contents of Register Rn (n=1,or 2) are stored in location X
5	TAD X,Rn	Two's complement add of location X contents and Rn stored in Rn(n=1, or 2)
6	CLEAR X	Set contents of location X to zero
7	CIR Rn	Complement and increment the register Rn(n=1, or 2)
8	SUB X,Rn	Subtract contents of X from register Rn(n=1, or 2)

9	DEC X	Subtract 1 from location X and store the results in X
A	INC X	Add 1 to contents of location X
B	CMP X	If contents of X>contents of R1, put 1 in R2 If contents of X<contents of R1, put 2 in R2 If contents of X=contents of R2, put 4 in R2
C	JMP X	Transfer control to memory location X
D	JMPGT X	Transfer control to memory location X if R2=1
E	JMPLT X	Transfer control to memory location X if R2=2
F	JMPEQ X	Transfer control to memory location X if R2=4

There are three pseudo-op mnemonics:

Pseudo op	Meaning
.BEGIN X	Place program at Location X (note X must be an address!)
.END	No more instructions follow
.DATA N	Create a 16-bit word. N is a signed decimal value that is converted to a 16-bit two's complement integer.

Every program will start with a .BEGIN and end with an .END pseudo op-code. There will be exactly one .BEGIN, and one .END pseudo op-code in a program. There are no blank lines in the input.

The spacing in the output produced by your program does not have to match the sample output exactly. However, data must be under the headings, and the heading verbiage must be exact. All columns in the output should be left justified. The symbol table listing should be in alphabetical order by symbol. All output should be uppercase.

Sample input:

```
.BEGIN 200
AGAIN:   LD NEG2,R1
         TAD FIVE,R1
         READ NUM
         SUB NUM,R1
         CIR R1
         CMP LEVEL
         JMPGT FINAL
         JMP AGAIN
FINAL:   ST R1, 1A6
         WRITE 1A6
         HLT
NUM:     .DATA 0
LEVEL:   .DATA 8
NEG2:    .DATA -2
FIVE:    .DATA 5
.END
```

Sample output:

LOCATION	OBJECT CODE	SOURCE
		.BEGIN 200
200	361A	AGAIN: LD NEG2,R1
202	561C	TAD FIVE,R1
204	1E16	READ NUM
206	8616	SUB NUM, R1
208	7400	CIR R1
20A	BE18	CMP LEVEL
20C	DE10	JMPGT FINAL
20E	CE00	JMP AGAIN
210	45A6	FINAL: ST R1, 1A6
212	2DA6	WRITE 1A6
214	0C00	HLT
216	0000	NUM: .DATA 0
218	0008	LEVEL: .DATA 8
21A	FFFE	NEG2: .DATA -2
21C	0005	FIVE: .DATA 5
		.END

SYMBOL	LOC
AGAIN	200
FINAL	210
FIVE	21C
LEVEL	218
NEG2	21A
NUM	216