ASU Annual Programming Competition 2007
Problem Set

Directions

Please read these directions carefully!

The following pages contain the problem set for the 2007 Arizona State University programming competition. There are ten (10) problems. You have four hours to solve these problems. Some problems are easier than others. You will receive the same credit for solving the easy problems as you will receive for solving the difficult ones, so it is in your best interest to read through all the problems before you begin solving.

All your solutions will need to read input from a file, and all output should be written to a file. The file names you should use are specified under the problem name. You may assume the input files exist in the directory in which your code is running; you should create output files in that same directory. Pay careful attention to the output-formatting instructions in the problem descriptions.

Good luck!

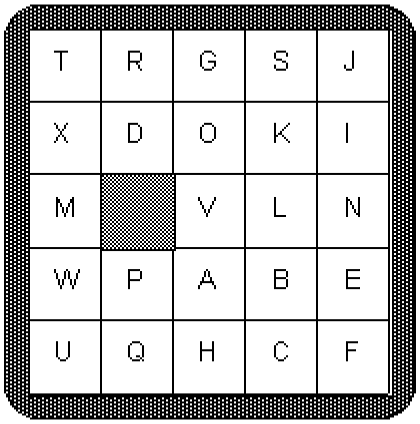
ASU Annual Programming Competition
2007

Problem A: Puzzle

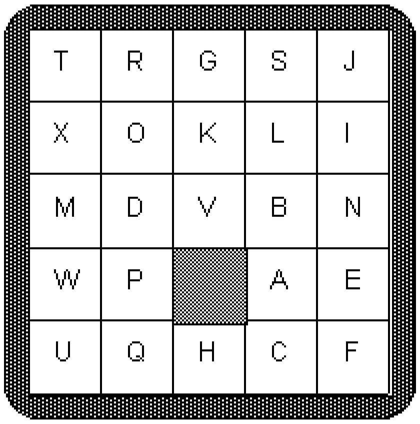
A children's puzzle that was popular 30 years ago consisted of a 5x5 frame, which contained 24 small squares of equal size. A unique letter of the alphabet was printed on each small square. Since there were only 24 squares within the frame, the frame also contained an empty position which was the same size as a small square. A square could be moved into that empty position if it were immediately to the right, to the left, above, or below the empty position. The objective of the puzzle was to slide squares into the empty position so that the frame displayed the letters in alphabetical order.

The illustration below represents a puzzle in its original configuration and its configuration after the following sequence of 6 moves:

1) `The square above the empty position moves.`
2) `The square to the right of the empty position moves.`
3) `The square to the right of the empty position moves.`
4) `The square below the empty position moves.`
5) `The square below the empty position moves.`
6) `The square to the left of the empty position moves.`



Original puzzle configuration.



Puzzle configuration after the sequence of described moves.

Write a program to display resulting frames given their initial configurations and sequences of moves.

Input

Input for your program consists of several puzzles. Each is described by its initial configuration and the sequence of moves on the puzzle. The first 5 lines of each puzzle description are the starting configuration. Subsequent lines give the sequence of moves.

The first line of the frame corresponds to the top line of squares in the puzzle. The other lines follow in order. The empty position in a frame is indicated by a blank. Each line contains exactly 5 characters, beginning with the character on the leftmost square (or a blank if the leftmost square is actually the empty frame position).

The sequence of moves is represented by a sequence of As, Bs, Rs, and Ls to denote which square moves into the empty position. **A** denotes that the square above the empty position moves; **B** denotes that the square below the empty position moves; **L** denotes that the square to the left of the empty position moves; **R** denotes that the square to the right of the empty position moves. It is possible that there is an illegal move, even when it is represented by one of the 4 move characters. If an illegal move occurs, the puzzle is considered to have no final configuration. This sequence of moves may be spread over several lines, but it always ends in the digit 0. The end of data is denoted by the character Z.

Output

Output for each puzzle begins with an appropriately labeled number (`Puzzle #1`, `Puzzle #2`, etc.). If the puzzle has no final configuration, then a message to that effect should follow. Otherwise the final configuration should be displayed.

Format each line for a final configuration so that there is a single blank character between two adjacent letters. Treat the empty square the same as a letter. For example, if the blank is in an interior position, then it will appear as a sequence of 3 blanks - one to separate it from the square to the left, one for the empty position itself, and one to separate it from the square to the right.

Separate outputs for different puzzles by one blank line.

Note: The first test case of the sample input corresponds to the puzzle illustrated above.

Sample Input

```
TRGSJ
XDOKI
M VLN
WPABE
UQHCF
ARRBBL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAA
LLLL0
ABCDE
FGHIJ
KLMNO
PQRS
TUVWX
AAAAABBRRRLL0
Z
```

Sample Output

```
Puzzle #1:
T R G S J
X O K L I
M D V B N
W P   A E
U Q H C F

Puzzle #2:
  A B C D
F G H I E
K L M N J
P Q R S O
T U V W X

Puzzle #3:
This puzzle has no final configuration.
```

ASU Annual Programming Competition

2007

Problem B: Marbles

I have some (say, n) marbles (small glass balls) and I am going to buy some boxes to store them. The boxes are of two types:

Type 1: each box costs c_1 cents and can hold exactly n_1 marbles
Type 2: each box costs c_2 cents and can hold exactly n_2 marbles

I want each of the used boxes to **be filled to its capacity** and also want to minimize the total cost of buying them. Since I find it difficult for me to figure out how to distribute my marbles among the boxes, I seek your help.

Input

The input file may contain multiple test cases. Each test case begins with a line containing an integer n (1 <= n <= 2,000,000,000), the total number of marbles. The second line contains c_1 and n_1, the cost and capacity of the box of type 1. The third line contains c_2 and n_2, the cost and capacity of the box of type 2. Here, c_1, c_2, n_1 and n_2 are all **positive** integers having values smaller than 2,000,000,000.

A test case containing 0 for n in the first line terminates the input.

Output

For each test case in the input, print a line containing the number of boxes of type 1 used, and the number of boxes of type 2 used, such that all the marbles are held, and the **total number of cents we have to pay is minimized.** If in a test case, the best strategy is to buy only boxes of type 1, then write 0 in the solution for the number of boxes of type 2 used. Since each used box has to be filled to its capacity, some test cases may not have solution at all. For those test cases which do have solutions, print out two **nonnegative** integers m_1 and m_2, where m_i is the number of boxes of t*ype i* required. For those test cases which have no solution, print "failed".

If a solution exists for a test case, the solution is unique.

Sample Input

```
43
1 3
2 4
40
5 9
5 12
50
4 2
12 5
0
```

Sample Output

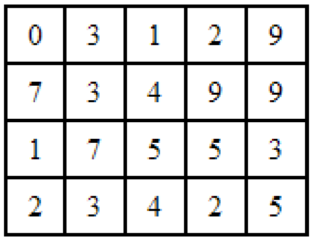
```
13 1
failed
25 0
```

ASU Annual Programming Competition
2007

Problem C: Number Maze

Input: maze.in
Output: maze.out

Consider a number maze represented as a two dimensional array of numbers comprehended between 0 and 9, as exemplified below. The maze can be traversed following any orthogonal direction (i.e., north, south, east and west). Considering that each cell represents a cost, then finding the minimum cost to travel the maze from one entry point to an exit point may pose you a reasonable challenge.



0	3	1	2	9
7	3	4	9	9
1	7	5	5	3
2	3	4	2	5

Your task is to find the minimum cost value to go from the top-left corner to the bottom-right corner of a given number maze of size *N* x *M* where *1 <= N, M <= 999*. Note that the solution for the given example is 24.

Input

The input file contains several mazes. The first input line contains a positive integer defining the number of mazes that follow. Each maze is defined by: one line with the number of rows, *N*; one line with the number of columns, *M*; and *N* lines, one per each row of the maze containing the maze numbers separated by one space.

Output

For each maze, output one line with the required minimum value to go from the top-left corner to the bottom-right corner.

Sample Input

```
2
4
5
0 3 1 2 9
7 3 4 9 9
1 7 5 5 3
2 3 4 2 5
1
6
0 1 2 3 4 5
```

Sample Output

```
24
15
```

Problem D: Parity

We define the parity of an integer **n** as the sum of the bits in its binary representation modulo 2. As an example, the number $21 = 10101_2$ has three *1s* in its binary representation, so it has parity *3 (mod 2)*.

In this problem you have to calculate the parity of an integer $\mathbf{1 \le I \le 2147483647}$.

Input

Each line of the input has an integer **I** and the end of the input is indicated by a line where **I = 0**, which should not be processed.

Output

For each integer **I** in the input, you should print a line "**The parity of B is P (mod 2).**", where **B** is the binary representation of **I**, and **P** is the sum of the bits in the binary representation.

Sample Input

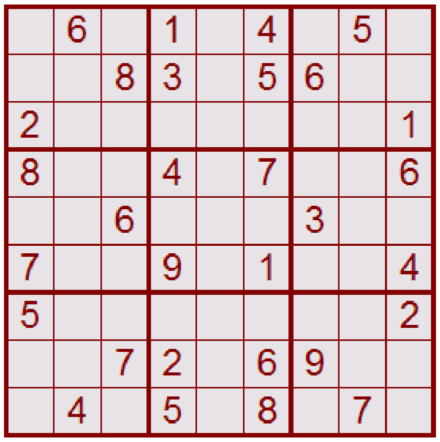
```
1
2
10
21
0
```

Sample Output

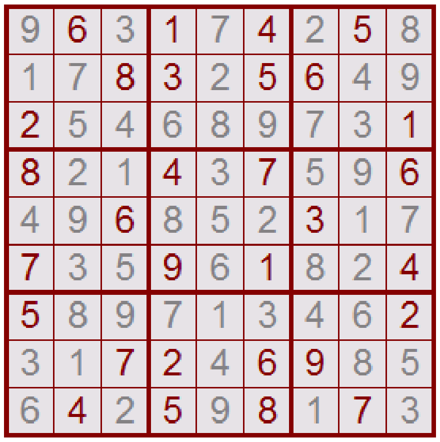
```
The parity of 1 is 1 (mod 2).
The parity of 10 is 1 (mod 2).
The parity of 1010 is 2 (mod 2).
The parity of 10101 is 3 (mod 2).
```

Problem E: Su DoKu

In many newspapers we may find some puzzles to solve, one of those is Su Doku. Given a grid *9×9* with some of entries filled, the objective is to fill in the grid so that every row, every column, and every *3×3* box contains the digits *1* through *9*.



	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	



9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

source: http://www.sudoku.com

Input

The first line of the input indicates the total number of test cases. Each subsequent test case contains an integer n such that $1 \leq n \leq 3$ and a grid $n^2 \times n^2$ with some of the entries filled with digits from *1* to n^2 (an entry not filled will have *0*). In this case, the objective is to fill in the grid so that every row, every column, and every $n \times n$ box contains the digits *1* through n^2. (No blank line between test cases)

Output

Output contains solutions for the problem. For each test case, if there exists more than one solution, you should give the lower one assuming a lexicographic order. For lexicographic comparison, you should consider lines in the first place. For example, if there are two solutions for a certain test case, and in the first line where they have difference, one solution begins with 5 2 3…, and another solution begins with 5 2 4…, you should give the first one as solution. If there is no solution, you should print '`NO SOLUTION`'. Print a blank line between test cases.

Sample Input

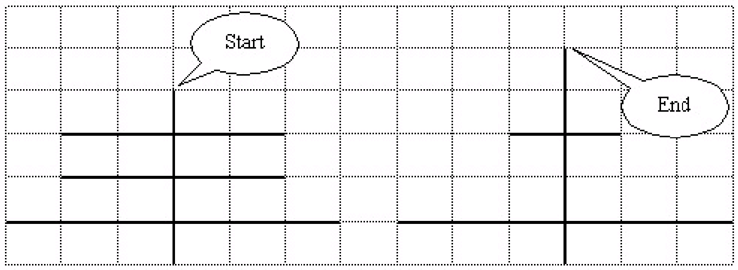
```
1
3
0 6 0 1 0 4 0 5 0
0 0 8 3 0 5 6 0 0
2 0 0 0 0 0 0 0 1
8 0 0 4 0 7 0 0 6
0 0 6 0 0 0 3 0 0
7 0 0 9 0 1 0 0 4
5 0 0 0 0 0 0 0 2
0 0 7 2 0 6 9 0 0
0 4 0 5 0 8 0 7 0
```

Sample Output

```
9 6 3 1 7 4 2 5 8
1 7 8 3 2 5 6 4 9
2 5 4 6 8 9 7 3 1
8 2 1 4 3 7 5 9 6
4 9 6 8 5 2 3 1 7
7 3 5 9 6 1 8 2 4
5 8 9 7 1 3 4 6 2
3 1 7 2 4 6 9 8 5
6 4 2 5 9 8 1 7 3
```

Problem F: Big Big Trees

There are **n** trees arranged in a straight line in the forest, the adjacent two trees are always **m** meters apart from each other. Each tree is **h** meters high (**h** may be different for different trees). At every integer height **i** **(1<=i<=h)**, the tree has two leaves with length **l_i** on both sides. The two leaves are left-right symmetrical, so each tree is left-right symmetrical too. The picture below shows two trees. Note that the longest leaf should be shorter than **m/2** meters. So every two leaves from different trees can NEVER overlap in left-to-right direction.



A monkey from the top of the first tree wants to reach the top of the last tree. He may climb up and down, and he may walk from left to right on the leaves or on the ground. He may also jump from the RIGHT ENDPOINT of a leaf to the LEFT ENDPOINT of another leaf on the next tree. Warning: the monkey cannot jump from or land inside a leaf or the ground! What's more, he jumps along a straight line, so the line should not contain any points of any other leaves, which means the jump line(segment, actually) cannot intersect with any other leaves different from the leaf he jumps from or lands to. Also he may jump only if the distance between the two endpoints is no longer than **k** meters. Clearly, the monkey should always use exactly **n-1** jumps, but he may walk on different route in order to minimize the total distance he walks (NOT climbs or jumps). Help him to calculate the minimal total distance walked.

Input

The first line of the input is a single integer **t(1<=t<=10)**, indicating the number of test cases. Each case begins with a line containing three integers **n, m, k** **(1<=n,m,k<=1000)**, indicating the number of trees, the distance between two adjacent trees, and the maximal allowed jump distance. There are **n** lines following. Each line describes a tree. The first integer **h(1<=h<=20)** is the height of the tree. There are **h** integers following: **l_1, l_2,... l_h** **($0<=l_i<m/2$, i=1,2...h)**, indicating the lengths of the leaves.

Output

For each case, print a number **d** in a single line indicating the minimal total distance the monkey walks.

Sample Input

```
2
2 7 3
4 3 2 2 0
5 3 0 1 0 0
3 50 40
4 15 3 16 10
8 12 12 12 21 12 15 6 14
13 15 23 20 18 14 1 21 9 9 18 23 10 4
```

Sample Output

```
5
28
```

Problem G: Ugly Numbers

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...

shows the first 11 ugly numbers. By convention, 1 is included.

Write a program to find and print the ugly numbers as required by the input.

Input

First line of the input contains a number *m*, which indicates the total number of ugly numbers we are interested in. *m* different numbers will appear in the subsequent lines. Each line contains a number *n*, which indicates that we are interested in the ***n*th** ugly number. 0<*n*<=150.

Output

The output should print out the ugly numbers we are interested in.
Each line for each interested ugly number.

Sample Input

```
3
4
7
10
```

Sample Output

```
The No.4 ugly number is 4.
The No.7 ugly number is 8.
The No.10 ugly number is 12.
```

ASU Annual Programming Competition
2007

Problem H: Wildcard

A wildcard ‘?’ in a string can either represent none character or any character from ‘a’ to ‘z’. Given two strings, which may have any number of ‘?’ in any place, if we can find a way to replace each ‘?’ either by none character or any character from ‘a’ to ‘z’, such that these two strings become identical, then we say that these two string match each other. Otherwise they don’t match.

For example, if string1 is “e???e??e” and string2 is “eaee”, these two strings match each other, since we can replace the first ‘?’ in string1 by ‘a’, and all the other ‘?’ by none character to make string1 become “eaee”. Another example is that string1 = “?” and string2 = “??a??”. These two strings match each other, since we can replace the first ‘?’ in string1 by ‘a’ and replace all the ‘?’ in string2 by none character to make them identical. An example of non-matching is given as follows: string1 = “eabce?” and string2 = “e??e??e”. In this problem, given any two strings, your task is to tell whether or not they match each other.

Input

The first line contains the number of test cases (*N*, 1=<N<=100). The subsequent 2*N* lines give the N test cases, two lines for each test case. Each line (except the first line, of course) contains a string, which may include any character from ‘a’ to ‘z’, and the wildcard ‘?’. The length of each string is L and 1=<L<=100.

Output

For each test case, if the two strings match each other, print out “match”, otherwise print out “not match”.

Sample Input

```
3
e???e??e
eaee
?
??a??
eabce?
e??e??e
```

Sample Output

```
match
match
not match
```

Problem I: Compound Words

You are to find all the two-word compound words in a given list. A two-word compound word in the given list is a word that is the concatenation of **exactly two** words in the same list. The two words can be either same or different. For example, if both words “so” and “soso” are in the given list, "soso" is a compound word. Another example, if the words “good”, “bye” and “goodbye” are in the given list, “goodbye” is a compound word.

Input

The input consists of a number of lowercase words, one per line, in alphabetical order. There will be no more than 120,000 words totally.

Output

Your output should contain all the compound words, which are the concatenation of **exactly two** words in the given list. Print one compound word per line, also in alphabetical order.

Sample Input

```
a
alien
born
less
lien
never
nevertheless
new
newborn
the
zebra
```

Sample Output

```
alien
newborn
```

Problem J: Train Swapping

At an old railway station, you may still encounter one of the last remaining "train swappers". A train swapper is an employee of the railroad, whose sole job is to rearrange the carriages of trains.

The title "train swapper" stems from the first person who performed this task at a railway bridge. The first train swapper had discovered that the bridge could be operated with at most two carriages on it. By rotating the bridge 180 degrees, the two carriages on the bridge switched the place, allowing him to rearrange the carriages (as a side effect, the carriages then faced the opposite direction, but train carriages can move either way, so who cares).

Now that almost all train swappers have died out, the railway company would like to automate their operation. Part of the program to be developed is a routine, which decides for a given train, the least number of swaps of two **adjacent** carriages necessary to rearrange all train carriages in the ascending order. Your task is to create that routine. Each carriage has a unique carriage id. You can assume that for a train which has *n* carriages, all the carriage ids are integers from 1 to n.

Input

The input contains the number of test cases (*N*) on the first line. Each test case consists of two input lines. The first line of a test case contains an integer *L*, determining the length (total number of carriages) of the train ($0<L<=50$). The second line of a test case contains a permutation of the numbers 1 through *L*, indicating the current order of the carriages. The carriages should be ordered such that carriage 1 comes first, then carriage 2, etc. and carriage *L* will come last.

Output

For each test case output the sentence: "`Optimal train swapping takes S swap(s).`", where *S* is an integer.

Sample Input

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

Sample Output

```
Optimal train swapping takes 4 swap(s).
Optimal train swapping takes 6 swap(s).
```