Problem A. Automat

Input file:	`automat.in`
Output file:	`automat.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

A *nondeterministic finite automaton without* ϵ*-transitions* is a directed graph, the nodes of which are called *states*, and the edges of which are called *transitions*. Each transition has an assigned *label*, which in this problem will always be either 0 or 1. Exactly one state is the *starting state*, and some subset of states, possibly including the starting state, is called the *final states*.

A *word* is a sequence of labels, in our case of 0s and 1s. An automaton is said to be *accepting* a given word if and only if there's a path from the starting state to one of the final states such that the labels in sequence along this path comprise the word.

Given a word of length n, find any nondeterministic finite automaton without ϵ-transitions that has at most $\lfloor \frac{n}{2} \rfloor + 1$ states such that it accepts the given word, but no other word of length n. $\lfloor x \rfloor$ denotes the largest integer not exceeding x.

Input

The first line of the input file contains the number of testcases t. Each of the next t lines contains one word of 0s and 1s. The length of each word is at least 1 and at most 100, the total length of all words in one input file is at most 10^5. All words in one input file are distinct.

Output

For each word, output the description of an automaton that accepts the given word but no other word of the same length, or a single line with -1 on it if there's no such automaton.

The first line of an automaton description should contain four numbers k, m, s and t, denoting the number of states, the number of transitions, the identifier of the starting state, and the number of final states. The states have identifiers from 1 to k. The second line should contain t distinct identifiers of the final states. The next m lines should contain the transitions, each transition described by three numbers: the identifier of the source state, the identifier of the destination state, and the label. There should be no duplicate transitions (but there might be two transitions that differ only by the label). k must not exceed $\lfloor \frac{len(word)}{2} \rfloor + 1$.

All numbers in each line should be separated by spaces.

Examples

automat.in	automat.out

2
0
01
	1 1 1 1
	1
	1 1 0
	2 2 1 1
	2
	1 2 0
	2 2 1

Problem B. Baum

Input file:	`baum.in`
Output file:	`baum.out`
Time limit:	3 seconds
Memory limit:	256 mebibytes

A *cycle* is an undirected graph with n vertices and n edges, such that there are edges between 1st and 2nd vertices, between 2nd and 3rd vertices, ..., between $(n-1)$-th and n-th vertices, and finally between n-th and 1st vertices.

Two players are dividing a cycle between them. The first player starts by taking an arbitrary vertex to himself. The second player can then take any other vertex to himself. They continue making alternating turns, taking one vertex that has not been taken before at each turn. At every turn except the first one, a player may only take a vertex that is connected by an edge to at least one vertex he has taken previously. The game ends when all vertices have been taken.

Each vertex has an associated positive integer value. Each player tries to maximize the total value of the vertices he takes. If both players play optimally, what will be the total value of each player's vertices in the end?

Input

The first line of the input file contains the integers n — the number of vertices, $3 \le n \le 10^6$. The second line of the input file contains n space-separated integers — the values of the vertices in the order along the cycle, each between 1 and 10^9.

Output

Output two integers separated by a space: the total value of the vertices that the player who moves first will get, and the total value of the vertices that the player who moves second will get.

Examples

baum.in	baum.out
4 1 2 3 4	5 5

Problem C. Entfernung

Input file:	`entfernung.in`
Output file:	`entfernung.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

Two famous superheroes Cheburator and Crocodile-man are being teleported to Manhattan to fight evil. The teleporter is a complex device, so it's not quite certain where exactly they will end up.

More precisely, we will represent Manhattan as a coordinate plane, and the area where they can end up in is a polygon on this plane. Inside this polygon, each superhero can end up in any point.

Even more precisely, the point where each superhero ends up is picked independently and uniformly at random. By *uniformly* we mean uniformity by area: for any figure inside the polygon, the probability to end up inside that figure is proportional to the area of that figure.

The success of the mission depends on the distance between the superheroes after they land. This being Manhattan, of course we're interested in the Manhattan distance: for two points (x_1, y_1) and (x_2, y_2), the distance is $|x_1 - x_2| + |y_1 - y_2|$.

What is the expected value of the distance between the superheroes?

Input

The first line of the input file contains one integer n, $1 \le n \le 1000$. The next n lines describe the vertices of the polygon in order (either clockwise or counterclockwise). Each vertex is described by two integers — its coordinates x and y, $-1000 \le x, y \le 1000$.

The polygon does not have self-intersections or self-touchings, but might be non-convex. The distance between any two points is still measured using the above formula without regard to the form of the polygon — in other words, the shortest "Manhattan path" might well pass partly outside the polygon.

Output

Output one floating-point number — the expected distance between the superheroes. Your output will be considered correct if it's within 10^{-8} absolute or relative error from the answer.

Examples

entfernung.in	entfernung.out

4
0 0
0 1
1 1
1 0
	0.6666666666666666

Problem D. Geld

Input file:	`geld.in`
Output file:	`geld.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

A *coin system* of order n is a set of positive integers not exceeding n — the denominations of coins. A given amount of money, a positive integer, is *representable* using the given coin system if it's possible to represent it as a sum of some denominations (possibly using one denomination multiple times).

For example, consider the coin system $\{3, 4\}$. It's not hard to see that all positive integers except 1, 2 and 5 are representable, but those are not.

A set of positive integers is called a *cost set* of order n, if there exists a coin system of order n such that all integers from the cost set are representable, and all other positive integers are not representable. For example, $\{3, 4, 6, 7, 8, 9, ...\}$ is a cost set of order 4, but not a cost set of order 3. Note that the empty set is a cost set of any order, corresponding to the empty coin system.

How many different cost sets of a given order exist?

Input

The first line of the input file contains the number of testcases t, $1 \le t \le 63$. Each of the next t lines contains one integer n, $1 \le n \le 63$ — the order of the cost sets to be counted.

Output

For each testcase, output the number of different cost sets of order n on a line by itself.

Examples

geld.in	geld.out
2 1 2	2 3

Problem E. Mannschaft

Input file:	`mannschaft.in`
Output file:	`mannschaft.out`
Time limit:	10 seconds
Memory limit:	256 mebibytes

When designing games, it's important to properly balance skill and randomness. If a higher skilled player always wins against a lower skilled player, the game will be too depressing. On the other hand, if the results are completely random, then there will be no satisfaction from winning the game.

In a newly designed game the skill of each player can be described by a floating-point number between 1 and 2, where 1 corresponds to a complete novice, and 2 to the best player ever. A player with skill s_1 will beat a player with skill s_2 with probability $\frac{s_1}{s_1+s_2}$ (there are no draws in this game).

Your team for this game consists of n players, with skills $a_1, a_2, \ldots, a_n$. The opposing team consists of m players, with skills $b_1, b_2, \ldots, b_m$. Each member of your team will play one game with each member of the opposing team, for a total of $n \times m$ games. What is the expected number of games won by each of your players?

Input

The first line of the input file contains two integers n and m, $1 \le n, m \le 10^6$. The second line contains n space-separated floating-point numbers between 1 and 2, with at most 8 digits after the decimal point, denoting the skills of your players. The third line contains m skills of the other team in the same format.

Output

Output n space-separated numbers: the expected number of games won for each of your players.

Your output will be considered correct if each number is within 10^{-8} absolute or relative error of the answer.

Examples

mannschaft.in	mannschaft.out

`3 2`
`1.0 2.0 1.5`
`1.0 2.0`
	`0.833333333333333 1.166666666666666`
	`1.028571428571428`

Problem F. Quadrat

Input file:	`quadrat.in`
Output file:	`quadrat.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

It is well-known that for any n there are exactly four n-digit numbers (including ones with leading zeros) that are *self-squares*: the last n digits of the square of such number are equal to the number itself. These four numbers are always suffixes of those four infinite sequences:

```
...0000000000
...0000000001
...8212890625
...1787109376
```

For example, $09376^2 = 87909376$, which ends with 09376.

You need to count the numbers that are *almost self-squares*: such that each of the last n digits of their square is at most d away from the corresponding digit of the number itself. Note that we consider digits 0 and 9 to be adjacent, so for example digits that are at most 3 away from digit 8 are 5, 6, 7, 8, 9, 0 and 1.

Input

The first line of the input file contains the number of testcases t, $1 \le t \le 72$. Each of the next t lines contains one testcase: two numbers n ($1 \le n \le 18$) and d ($0 \le d \le 3$).

Output

For each testcase, output the number of almost self-squares with length n and the (circular) distance in each digit from the square at most d on a line by itself.

Examples

quadrat.in	quadrat.out

2
5 0
2 1
	4
	12

Note

In the second example case, the 12 almost self-squares are: 00, 01, 10, 11, 15, 25, 35, 66, 76, 86, 90, 91.

Problem G. Sprache

Input file:	`sprache.in`
Output file:	`sprache.out`
Time limit:	3.5 seconds
Memory limit:	256 mebibytes

You've decided to create a new language which will have very simple rules. Whenever we need a word for a new concept, we will just generate it randomly!

The language will use the same 26 letters English uses. Each letter is either a consonant or a vowel, but these are not necessarily the same as in English — the only thing you know is that at least one letter is a consonant, and at least one letter is a vowel.

When we need to generate a new word, we will pick the first letter uniformly randomly from all 26 letters. When picking each consecutive letter, we will discourage long sequences of vowels and long sequences of consonants. More specifically, if the last k letters have all been vowels, the relative probability to pick each vowel as the next letter will be $\left(\frac{1}{2}\right)^k$ of the relative probability to pick each consonant. The consonants have a more relaxed rule: if the last k letters have all been consonants, the relative probability to pick each consonant will be $\left(\frac{2}{3}\right)^k$ of the relative probability to pick each vowel.

For example, suppose letters `a`, `e`, `i`, `o`, `u` are vowels, and all remaining letters are consonants, and suppose we've already generated a word '`sprach`'. The last two letters are consonants, so consonants have a relative probability of $\frac{4}{9}$ compared to vowels. There are 21 consonants and 5 vowels, so the probability of picking each particular consonant as the next letter is $\frac{\frac{4}{9}}{21\cdot\frac{4}{9}+5} = \frac{4}{129}$, and the probability of picking each particular vowel as the next letter is $\frac{1}{21\cdot\frac{4}{9}+5} = \frac{9}{129}$.

Given a word in such language, determine which letters are vowels and which letters are consonants.

Input

The testcases for this problem are generated randomly. The first line of the input file contains one integer n, the number of words to determine the vowels and consonants for. In all inputs except the sample input $n = 100$.

The next n lines contain one word each. In all inputs except the sample input each word will have length 1000. Each word contains only lowercase English letters. Each word is generated randomly and independently according to the following procedure:

1. Pick which letters are vowels and which letters are consonants uniformly out of all $2^{26} - 2$ possible assignments with at least one vowel and at least one consonant.
2. Generate a word of length 1000 according to the algorithm described in the problem statement.

There are 10 non-sample inputs in this problem.

Output

Output n lines. For each of the input words, output one line with 26 letters. i-th letter of this line should be '`V`' if i-th letter of the English alphabet is a vowel for that word, and '`C`' if it's a consonant for that word.

Your answer will be considered correct if for at least 95 words out of 100 all 26 letter types will match the types used for generating the corresponding word. Any correctly formatted output will be accepted for the sample input.

Examples

sprache.in	sprache.out
2 just example	VCVCVCVCVCVCVCVCVCVCVCVCVC CCCCCCCCCCCCCCCCCCCCCCCCCV

Problem H. Steigung

Input file:	`steigung.in`
Output file:	`steigung.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

You've just started your research into *trigonometric neural networks*. As the first step, you need to learn to compute gradients of complex trigonometric functions.

More precisely, you have an expression that contains 26 variables, denoted by lowercase English letters, and two operations: addition and sine. For example: `sin(x+y+sin(z+t))`. It contains no extra parentheses except the ones enclosing the arguments of a sine operation.

Given such expression $func$, and the values a_0, b_0, …, z_0 for all variables, compute the partial derivatives over each variable. For example, for variable p, compute

$$\lim_{p \to p_0} \frac{func(a_0, b_0, ..., o_0, p, q_0, ..., z_0) - func(a_0, b_0, ..., o_0, p_0, q_0, ..., z_0)}{p - p_0}$$

Input

The first line of the input file contains the expression with sines and additions, as described above, without whitespace. The length of the expression is at most 300. The second line of the input file contains 26 space-separated floating-point numbers, each between -10 and 10, with at most 8 digits after the decimal point — the values a_0, b_0, …, z_0.

Output

Output 26 space-separated floating-point numbers, denoting the partial derivatives of the given expression over each variable in the given point. Your output will be considered correct if each number is within 10^{-8} absolute or relative error of the answer.

Examples

steigung.in	steigung.out

sin(x+x+y)+sin(z)
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.23
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 0.000000000000000
	0.000000000000000 2.000000000000000
	1.000000000000000 0.973666395005375

Problem I. Vieleck

Input file:	`vieleck.in`
Output file:	`vieleck.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

In order to generate random testcases for problem "Entfernung", we need a way to randomly generate polygons which are not necessarily convex (but of course without self-intersections and self-touchings).

At first, we've employed the following strategy to generate a polygon with n vertices:

1. Generate n independent points by picking each of $2n$ coordinates independently and uniformly at random from integers between -1000 and 1000.
2. In case any two points coincide or any three points are on the same line, go back to step 1.
3. Generate a random permutation of n points uniformly (each permutation is picked with the same probability). Connect those n points into a polygon in the order of the permutation, also connecting the last element of the permutation with the first element of the permutation.
4. In case the resulting polygon has self-intersections, go back to step 3, otherwise we're done!

After trying out this strategy for a while, we've noticed that it can take a really long time even for moderate values of n, because self-intersections are actually very likely when we connect the vertices in the order of a random permutation.

We've decided to improve the strategy as follows:

1. As before.
2. As before.
3. As before.
4. In case the resulting polygon does't have any self-intersections, we're done!
5. Otherwise, pick any pair of intersecting edges, uniformly at random from all such pairs. Remove those two edges. Connect the four loose ends with two edges in such a way that we still have one polygon, and it's different from the one we just had (there's exactly one such way), then go back to step 4.

It's possible to prove that this strategy always terminates, and in fact it terminates much faster than the first strategy.

However, it turns out that it tends to generate somewhat different polygons. Given a polygon, determine which strategy was used to generate it.

Input

The first line of the input file contains one integer t, the number of testcases. Each testcase starts with the number of vertices n on a line by itself, followed by n lines, each containing the coordinates of a vertex.

In non-sample inputs one of the two strategies was picked uniformly at random for each testcase, and the polygon was generated using the picked strategy.

In non-sample inputs t is always 1000 and n is always 12.

There are 10 non-sample inputs in this problem.

Output

Output t lines, each containing either the word "`FIRST`" (without quotes), or the word "`SECOND`" (without quotes) denoting the strategy picked to generate the corresponding polygon. Your output for non-sample inputs will be accepted if at least 666 out of 1000 strategies are guessed correctly. Any correctly formatted output will be accepted for the sample input.

Examples

vieleck.in	vieleck.out
2	SECOND
4	FIRST
0 0	
0 10	
10 10	
10 0	
4	
0 0	
10 0	
1 1	
0 10	

Problem J. Zubereitung

Input file:	`zubereitung.in`
Output file:	`zubereitung.out`
Time limit:	2 seconds
Memory limit:	256 mebibytes

Andrew is preparing problems for his next contest in Petrozavodsk. He has two types of problems: easy problems take t_1 minutes to prepare each, and hard problems take t_2 minutes to prepare each ($t_1 < t_2$). Each day Andrew has d minutes to spend on preparing problems.

Andrew has decided on the following algorithm for picking the order in which he prepares the problems. First, he arranges all problems he has to prepare in a fixed sequence. Each day, he goes through this sequence from left to right, and whenever he encounters a problem that is not yet prepared but can be prepared today, he prepares it, then continues looking through the sequence. Note that he does not stop looking if he encounters a problem that he doesn't have enough time to prepare — maybe another problem further down the sequence requires less time. On the next day, he starts looking at the sequence again from the very beginning (of course, skipping the problems that he has already prepared), and so on until all problems are prepared.

Andrew is wondering: is this algorithm optimal? More precisely, do there exist two sequences of problems, which contain the same set of problems (in other words, the same amount of easy problems and the same amount of hard problems), but require different number of days to prepare if Andrew follows his algorithm?

Input

The first line of the input file contains the number of testcases t, $1 \le t \le 1000$. The next t lines contain one testcase each, described by three integers t_1, t_2 and d, $1 \le t_1 < t_2 \le d \le 100$. All testcases in one input file are different.

Output

For each testcase, output two sequences of problems, one per line, that make Andrew's algorithm spend different number of days, but have the same set of problems. Each sequence should be printed without spaces, with character '`E`' denoting an easy problem and character '`H`' denoting a hard problem. The length of each sequence must be at most 10000. It is guaranteed that if a solution exists, there exists a solution where each sequence has length at most 10000. If there's no solution for a particular testcase, print "`OPTIMAL`" (without quotes) on one line.

Examples

zubereitung.in	zubereitung.out

2
1 2 2
1 2 3
	OPTIMAL
	EEEHHH
	HHHEEE

Note

In the second example case, ordering `EEEHHH` needs four days to prepare: three easy problems in the first day, and one hard problem in each of the following days. At the same time, `HHHEEE` has the same set of problems and requires just 3 days: every day Andrew would start with a hard problem, and then skip over remaining hard problems until the next easy problem, so he would prepare a hard problem and an easy problem each day.