IPSC 2012

problems and sample solutions

Problem P: Partitioning containers

Our company is shipping cargo with total weight s. The shipping company prepared two ships. Both have a maximum capacity of $s/2$. Unfortunatelly our cargo is already packed into n containers of various weights. We are not able to open the containers, so we need to decide which containers should go on which ship.

That sounds as a hard problem (especially to computer scientists). Luckily, our insurance policy makes it possible to skip shipping *one* container.

Problem specification

You are given n positive integers with sum s. Erase at most one of them, and split the remaining ones into two groups. The sum of numbers in each group must not exceed $s/2$.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of two lines. The first line contains the number n of containers. The second line contains n positive integers: the weights of containers. The containers are labeled 1 through n in the order in which they appear in the input.

In the easy data set all test cases have $n \leq 20$. In the hard data set we have $n \leq 15\,000$.

Output specification

For each test case output two lines. Each line should describe the cargo of one ship. The first number in the line must be the number of containers in the ship's cargo. The rest of the line should contain the labels of all containers carried by the ship, in any order. Any valid solution will be accepted.

Example

input

```
2

4
10 20 30 71

4
10 10 10 10
```

output

```
1 1
2 2 3
2 1 4
2 3 2
```

First test case:
We have four containers with weights 10, 20, 30, and 71. Therefore $s = 10 + 20 + 30 + 71 = 131$, *and each ship will carry at most* $131/2 = 65.5$. *In the solution described above we put container #1 on the first ship, containers #2 and #3 on the second ship, and we throw away container #4. (Another valid solution is to put the first three containers on the same ship and to leave the other ship empty.)*

Second test case:
Note that we are not required *to throw away one of the containers. Sometimes we may be able to ship all of them. (But of course, there are other valid solutions that only ship three of the four containers.)*

Task authors

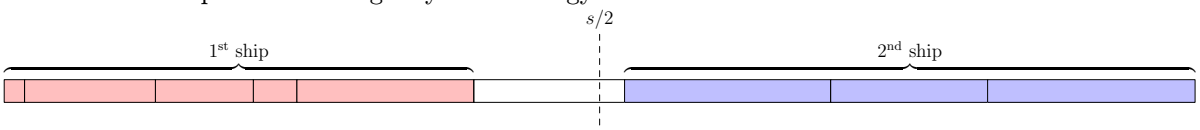
Problemsetter:	Vlado 'Usamec' Boža
Task preparation:	Vlado 'Usamec' Boža, Monika Steinová

Solution

There are many ways how to solve this problem. One of the most straightforward approaches looks like this: Process the containers one at a time, in any order. Load them onto the first ship as long as they fit. Once you get the first container that does not fit, throw it away. Load all remaining containers onto the second ship.

Proof: Let the containers' weights be $a_1, \ldots, a_n$. Let a_x be the first container that does not fit onto the first ship. This means that $a_1 + \cdots + a_x > s/2$. Thus $a_{x+1} + \cdots + a_n < s/2$, which means that the remaining items indeed do fit onto the second ship.

Here is a nice picture showing why this strategy works:



Another strategy that works: Throw away the largest container. (This alone clearly cannot hurt you.) For each other container, try putting it onto the first ship, and if it does not fit, put it onto the second ship.

Why does this strategy work? Let x be the weight of the heaviest container (the one you discarded) and let y be he weight of an item you are later trying to send. Immediately before processing y the total remaining carrying capacity of both ships is at least $x + y$. Thus one of the ships can still carry an item that weighs $(x + y)/2$. But as $x \geq y$, we have that $(x + y)/2 \geq y$, q.e.d.

There are many other strategies that work. On the other hand, there are also many *incorrect* greedy strategies for this problem. For example, here's one that does not work:

Process the containers one at a time, in any order. In phase one, you load the containers onto the first ship. Once you get a container that does not fit, move to phase two. In phase two you try to load the containers onto the second ship, and if you get one that does not fit, you throw it away.

Can you find a counterexample to this strategy?

And what if we start by sorting the containers in descending order?

Problem Q: Quaint encryption

Bob and Alice are constantly sending notes to each other during school lessons. As they sit on opposite sides of the classroom, the notes have to be handed over via several of their classmates.

One of those classmates is Mike. He is bored and wants to read the notes. Unfortunately for him, Bob and Alice are encrypting the messages, and he was unable to crack their code.

Then, one day Mike got a lucky break: Bob dropped a paper with decrypted words from one of Alice's messages. The words were not in their proper order, but this still helped Mike to find the encryption method. Afterwards, he could easily read all their messages.

Problem specification

Alice and Bob have a very simple encryption method: the substitution cipher, using just the lowercase letters of the English alphabet. The key to the cipher is some *permutation* of letters – a string that contains each of the 26 letters exactly once. For example, one such permutation is `ebmhcdfgijklnoapqrswtuvxzy`.

To encode a message, simply replace the i-th letter of the alphabet by the i-th letter of the key, for all i. For example, if we use the above key to encode a message, all '`a`'s will be encoded as '`e`'s, all '`b`'s as '`b`'s, all '`c`'s as '`m`'s, and so on. The word "`beef`" would then be encoded as "`bccd`".

You will be given a list of original words, and a list of the same words after they have been encrypted. The words in the second list are **not necessarily** in the same order as the words in the first list.

Your task is to produce any key that encodes the first set of words into the second set of words.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case starts with an integer n – the number of words in each list. Each of the following n lines contains one original word. Afterwards there are another n lines with the encrypted words. All words use lowercase English letters only. In the easy data set $n \leq 30$, in the hard data set $n \leq 10\,000$. No word has more than 30 characters.

Output specification

For each test case, output a single line containing a permutation of the 26 lowercase letters.

You may assume that for each test case at least one valid permutation exists. If there are multiple valid solutions, you may output any of them.

Example

input

```
1

2
dogg
cat
mew
haff
```

output

```
ebmhcdfgijklnoapqrswtuvxzy
```

In this case, the original words are `dogg` *and* `cat`*. Clearly, the encrypted form of* `dogg` *is* `haff` *and the encrypted form of* `cat` *is* `mew`*. This tells us that any permutation of the form* `e.mh..f.......a....w.....` *will be a good encryption key. The example output contains one such permutation.*

Task authors

Problemsetter:	Martin 'rejdi' Rejda
Task preparation:	Martin 'rejdi' Rejda, Jano Hozza

Solution

This problem is known as the *known plaintext attack* – you have an open text and its encrypted form, and your goal is to find the encryption key.

There are 26! possible encryption keys, so trying all of them is clearly out of the question. We have to reduce the search space somehow. Here are some ideas to get you started:

Word lengths. As in the example in the problem statement, we may note that the encryption preserves word lengths. We may pick an unencrypted word and try pairing it with an encrypted word of the same length. This tells us something about the key. We then check whether this partial key looks plausible, possibly deducing more and more about it. And if the check fails, we try a different encrypted word.

Statistical analysis. Probably the simplest way of solving most of the test cases is to look at letter frequencies. The most frequent letter in the first set of words has to be encrypted to the most frequent letter in the second set of words. The same holds for the second most frequent letter, and so on. Of course, there may still be some ties. In that case we may get more information by looking at pairs or triplets of consecutive letters.

The test cases we used for this problem were not too tricky, so any reasonable approach had a good chance of solving almost all of them. And if you get stuck, remember that it is perfectly valid to solve some of the test cases (partially) by hand.

Our implementation that solves all the test cases used in the contest uses letter frequencies to reduce the set of possible options, and then tries pairing the words until a correct solution is found.

Problem R: Rebel Alliance attacks

Oh no! The Rebel Alliance have found plans for a second Death Star – the deadliest weapon in the universe. It can destroy entire planets in a few seconds. If they allow it to be unleashed, it will mean a certain defeat.

The mission is clear – find Death Star and destroy it in a single big explosion. But now the Alliance needs to know how many explosives they need to prepare. And thus they need your help.

The Death Star can be represented as a perfect sphere with origin $(0, 0, 0)$ and radius r. Rebel scientists have determined that to ensure its destruction, the Rebels must infiltrate it and place explosives at all points (x, y, z) inside the sphere such that x, y, z are integers, and the distance between $(0, 0, 0)$ and (x, y, z) is also an integer.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consist of a single line containing a single integer r – the radius of the Death Star.

The largest test cases have $r = 100$ in the easy data set and $r = 44\,444$ in the hard data set.

Output specification

For each test case, output a single line containing a single integer – the number of explosives needed.

Example

input

```
3

1

2

5
```

output

```
1
7
49
```

In the first test case, the only explosive needs to be placed at the centre of the Death Star. In the second test case, additional explosives must be placed at positions $(\pm 1, 0, 0)$, $(0, \pm 1, 0)$, $(0, 0, \pm 1)$.

Task authors

Problemsetter: Peter 'ppershing' Perešíni
Task preparation: Peter 'ppershing' Perešíni, Michal 'mišof' Forišek

Solution

The tasks is to count all tuples (x, y, z, w) such that $x^2 + y^2 + z^2 = w^2$ and $w < r$.

Easy data set

The easiest way how to solve this task is just to check all tuples where $-r < x, y, z < r$ and $0 \leq w < r$. The time complexity of this approach is $O(r^4)$. It should solve the small data set in less than a minute, so it may be a good strategy to just implement this simplest solution. Alternately, it is not too hard to write a solution that solves the easy input faster. Here are two basic improvements we can make:

Exploit symmetry. Instead of iterating over the range $(-r, r)$, we can exploit the fact that $x^2 = (-x)^2$. Thus, it is sufficient to iterate just over the range $[0, r)$. Whenever we find a solution where $x \neq 0$, we know that this represents two solutions to the original problem. This can easily be generalized to multiple variables. In that case the tuple (x, y, z, w) with $x, y, z \geq 0$ represents $2^{[x\neq 0]+[y\neq 0]+[z\neq 0]}$ solutions of the original problem.

Calculate the fourth value. Instead of "guessing" the value of r by iterating over all possibilities, one can just calculate $r = \sqrt{x^2 + y^2 + z^2}$ and check whether the resulting number is an integer. Note that this gives us an $O(r^3)$ solution – still not fast enough, but already a huge improvement.

Hard data set

For the hard data set we need a much faster solution. One good starting point towards such a solution is to notice that the equation $x^2 + y^2 + z^2 = w^2$ can be rewritten as $x^2 + y^2 = w^2 - z^2$.

To count all solutions to this equation, we can use a meet-in-the-middle approach: Let a_t be the number of solutions for the equation $x^2 + y^2 = t$, and let b_t be the number of solutions for the equation $w^2 - z^2 = t$. The total number of solutions to the original equation can then be computed as $\sum_t a_t \cdot b_t$. (And it is obviously sufficient to limit the summation to the range $t \in [0, r^2)$.)

This gives us a solution to the hard data set with time complexity $O(r^2)$, which should be fast enough. Note that the answer for the largest test case overflows a 32-bit integer.

Not enough memory

And it is fast enough – provided your computer has enough RAM.

The solution above relied on the fact that we can precompute (at least) all the values a_t. The problem is that storing them requires a lot of memory. The largest test case has $r = 44\,444$. If we use a simple array to store the values a_t, even at two bytes per entry we would still require 4 GB. Hashmaps don't help too much either – if r is on the order of 10 000, the number of non-zero a_t is roughly $r^2/10$. Of course, both options are usable if you have enough memory, or if you are willing to wait while your computer uses the swap like crazy.

Alternately, we can try to reduce the memory usage. We know at least two possible approaches. Both are based on the idea of dividing the computation of a_t and b_t into several disjoint classes, solving each class separately and them summing the results. There are two natural types of such classes one can exploit:

Successive ranges. A natural trick is to split the computation into several consecutive ranges of length s. We will make several independent passes. In the i-th pass, we will only store the values a_t and b_t for $t \in [s(i-1), si)$. A reasonable choice is for example to take $s = 10^6$.

It would seem that this approach is r^2/s times slower than the previous solution. However, we do not have to try all pairs (x, y) in each iteration, we can dynamically compute lower and upper bounds on x and y to keep $x^2 + y^2$ within the current range.

The other option is to use **equivalence classes** modulo some m. The main observation here is that $x^2 + y^2 \equiv (x + km)^2 + (y + lm)^2 \pmod{m}$ for any integers k, l. The same holds for subtraction.

The actual solution: In the first phase we consider the pairs (x, y) with $0 \le x, y < m$. For each of them, we compute $\varphi_{x,y} = (x^2 + y^2) \bmod m$ and store the pair (x, y) into the bucket $\varphi_{x,y}$. We do the same with the pairs (z, w).

In the second phase, we consider each each remainder modulo m separately. For each remainder ξ, we take all the pairs inside the bucket ξ and use them to generate all pairs (x, y) with $0 \le x, y < r$ and $\xi = (x^2 + y^2) \bmod m$. In this way we will compute all values a_t for $t \equiv \xi \pmod{m}$. We do the same with the other buckets to compute the corresponding values b_t.

By selecting $m = 512$ we can get a good balance between memory and speed. Note that in this case, the compiler can actually convert the expensive modulo operation into a fast bitwise or.

Beyond the hard data set

There are even faster solutions. For example, the Online Encyclopedia of Integer Sequences contains the sequence http://oeis.org/A016725 that gives, for each w, the number of solutions $a(w)$ to the equation $x^2 + y^2 + z^2 = w^2$. Clearly, the answer to our problem is then simply $\sum_{w<r} a(w)$.

We have $a(0) = 1$ and $\forall n > 0 : a(n) = 6b(n)$, where b is a function with the following properties:

- $b(2^i) = 1$ for any $i > 0$
- $b(p^i) = p^i$ for any $i > 0$ and any prime p such that $p \equiv 1 \pmod{4}$.
- $b(p^i) = p^i + 2(p^i - 1)/(p - 1)$ for any $i > 0$ and any prime p such that $p \equiv 3 \pmod{4}$.
- Function b is multiplicative: we have $b(xy) = b(x)b(y)$ whenever x and y are relatively prime.

We can compute the values $b(1)$ through $b(r - 1)$ all at the same time using a slightly modified version of the Sieve of Eratosthenes. The proof of the formula is left as an exercise to the reader :-) (or follow the links from the OEIS page).

Problem A: Abundance of sweets

Georg has a new profession: he now works in a candy factory. The job is way more boring than it sounds: he is responsible for quality assurance. And that means taking a box of candies, counting them and checking that none are missing. Help poor Georg! Write a program that will do the job for him.

You will be given a matrix with r rows and c columns. This matrix represents the top view of a box of candies. The matrix will only contain the following characters:

- "`.`": a free spot
- "`o`": the edible part of the candy
- "`<>v^`": candy wrapper

There are exactly two ways how a whole piece of candy looks like:

```
1. >o<    2. v
             o
             ^
```

Whenever you see three characters arranged in this way, you see a whole piece of candy.

Problem specification

For the given matrix count the number of whole pieces of candy it contains.

In the **easy data set** the box will only contain whole pieces of candy and nothing else.

In the **hard data set** the box can also contain fragment of candies: characters `o<>v^` that don't belong to any whole candy. To make your life easier, you may assume that the following configuration will never appear in the hard data set:

```
 v
>o<
 ^
```

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

The first line of each test case contains two integers r and c ($1 \leq r, c \leq 400$): the dimensions of the matrix. Each of the next r lines contains one row of the matrix: c characters from the set "`.o<>v^`". (Their ASCII values are 46, 111, 60, 62, 118, and 94.)

Output specification

For each test case, output a single line with one integer – the number of whole candies in the box.

Example

input

```
1

5 4
.>o<
v.^.
ooo.
^.^.
>o<<
```

output

```
3
```

There are three whole candies: in the first row, in the last row, and in the first column.

Task authors

Problemsetter:	Michal 'Žaba' Anderle
Task preparation:	Michal 'Žaba' Anderle, Tomi Belan

Solution

Problem A was, as usual, the easiest task of the contest.

For the **hard data set** we need to traverse the input twice: once for each row and once for each column, checking for horizontal and vertical candy, respectively. Alternately, we traverse it just once, and whenever we see an "o", we check whether it forms a whole piece of candy with its neighborhood. So if it has ">" on its left and "<" on its right, or "v" above and "^" below, it is a candy.

The **easy data set** had an even simpler solution. As we have the guarantee that the box only contains whole pieces of candy, we know that each symbol "o" represents one candy. So all we need is to count the "o"s in the box.

Problem B:

(This page is intentionally left blank.)

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Tomi Belan, Michal 'mišof' Forišek, Michal 'Žaba' Anderle, Jano Hozza

Solution

Who needs a problem statement? Real programmers just submit immediately. And if they don't get "Accepted" straight out of the box, they just tweak the submission until it's OK.

(Fun fact: During the contest 20 teams asked us where to find the problem statement.)

The easy subproblem

Once you decided to just go ahead and submit something, you probably received the following answer:

```
Wrong answer: Not a sequence of positive integers
(Note: You may make at most 30 submissions for this subproblem.)
```

Okay. Clearly it is possible to submit this problem after all! And even better, it tells us that we should submit a sequence of positive integers. So let's try submitting a sequence, for example "1 2 3". Hooray, a new wrong answer: "`Sum of the sequence is not 123456789`". So now we need a sequence with this sum. If you submit a few such sequences, you may see the errors "`Sequence contains less than 6 numbers`" and "`Element #2 must be smaller than element #3`". We already got quite far – we are looking for a long-ish strictly increasing sequence with sum equal to the given magic number. Now comes the last type of error messages: "`Element #3 must be a multiple of element #2`".

Any sequence with length at least 6 and sum 123456789, in which each element is a proper divisor of the next element, would actually be accepted. The simplest way to produce such a sequence is to write 123456789 in base-2 and to output the powers of 2 that sum to 123456789.

The hard subproblem

In this subproblem we already know what to do – well, at least in general. The start is easy: Your first error message was almost certainly "`Not a string of six digits (0-9)`". But that's where the going gets tougher – the grader stops being too helpful. For almost any string of digits you would just see a message like "`"000047" is not the correct answer (badness 123)`". As you change your submission, the badness changes as well.

An educated guess in this situation is that the badness measures how far your guess is from the correct answer. Minor tweaks to the submitted number usually seem to support this theory. For instance, you could have seen a pattern like this one: `000000` has badness 123, `000001` has badness 120, `000002` has badness 119, and `000003` has badness 120 again.

Intuitively, we are trying to make changes that decrease the badness. With a good intuition this approach can actually work, and you should get to the correct answer within the 30 allowed submissions. The more you can observe about the patterns how badness changes, the better for you – the fewer submissions you'll need.

It is also not that hard to find out precisely what is going on: Each guess represents a point in 6-dimensional space, and the badness is simply the squared Euclidean distance from the correct point. Formally, if the correct answer is $a_1a_2a_3a_4a_5a_6$ and your guess is $b_1b_2b_3b_4b_5b_6$, the reported badness will be $(a_1 - b_1)^2 + \cdots + (a_6 - b_6)^2$. The easiest way to come up with this idea is to notice that if you change a single digit, badness is a quadratic function.

Once you make this observation, you probably already have enough information to compute the correct answer. In the worst case you'll still need several random submissions. (Most sets of five random guesses have the property that you can determine the correct answer with 100% certainty.)

Problem C: Card game

You are enjoying a lovely weekend afternoon in a new amusement park. The park also offers a variety of activities that are about winning or losing a little money. One such activity has just caught your attention: a simple card game. As a person interested in puzzles and riddles, you almost instantly started to wonder: Should I pay for such a game? Am I expected to win or lose money while playing it?

This game is played with a standard deck of cards – there are four suits and in each suit cards worth 2 through 10, a Jack, a Queen, a King and an Ace. For the purpose of this problem, the Aces have value 1, Jacks 11, Queens 12, and Kings 13. In other words, each of the values 1 through 13 is present four times in the deck.

The simplest version of this game looks as follows: An employee of the amusement park shuffles the deck uniformly at random and gives you a card. Your winnings are equal to the value of the card.

Here is a more complicated version: As before, the dealer gives you a random card. You now have two options: either you take your winnings, or you return the card you have and ask for another one. The second one is again drawn uniformly at random, and you have to keep it.

The general version of this game has x turns, and you know x in advance. In each turn you can keep the card you have and end the game. In each turn except for the last one you can return the card and ask for another one.

In the **easy data set** the card you return is always returned back to the deck. Thus the new card you then get is always picked uniformly at random from a deck of 52 cards. In the **hard data set** the card you return is thrown away, so the more turns you take, the smaller the deck becomes.

Problem specification

You are given the maximal number of turns x. Find the maximum expected amount of money you can win in a single game.

(In other words, find the smallest y with the property: if playing a single game costed more than y, the amusement park would still profit from the game, even if all players played it optimally.)

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case is given as a single line with a single integer x (between 1 and 52, inclusive).

Output specification

For each test case, output a single line with one real number – the expected amount you will win if you follow an optimal strategy. Output at least 10 decimal places. Any answer with an absolute or relative error at most 10^{-9} will be accepted.

Example

input	output
1 1	7.000000000000

This is the correct answer both for the easy and the hard version, as for $x = 1$ they are the same: you have to keep the first card you get. From the 52 cards, each card is picked with probability $1/52$. There are 4 cards having each of the values 1 through 13. Hence the expected value of a card is $\sum_{i=1}^{13} i \cdot (4/52) = 7$.

Task authors

Problemsetter: Michal 'mišof' Forišek
Task preparation: Monika Steinová, Michal 'mišof' Forišek

Solution

This problem is based on a traditional brain teaser. We were unable to determine its original author. If you know something, let us know!

The easy subproblem

First note that when we pick a single card uniformly at random from the deck of 52 cards, the expected value of that card is 7. (This is just the average value of all cards in the deck.)

The task can now be solved backwards. Before drawing your last card your expected result is 7. Now consider that you are in the previous turn. You hold a card in your hand and you ask yourself the question: should I keep it, or should I swap it for a random card? Well, now that you know the expected profit from a random card, the answer is clear: return any card with value less than (or possibly equal to) 7.

What is then your expected profit for a game with two turns? In the first draw with probability 7/13 you will get a card that costs at most 7, and you swap it for an expected profit of 7. For each x between 8 and 13, inclusive, with probability 1/13 you get a card that costs x in the first draw and you keep it. Thus the expected profit is $(7/13) \cdot 7 + 8/13 + 9/13 + \cdots + 13/13 = 112/13 \approx 8.615$.

Now we see the optimal strategy for the first turn of a three-turn game: if your card costs more than 8.615, keep it, if it costs less, throw it away – the expected profit from playing on is greater than what you have now. And we continue in the same fashion for larger x.

The hard subproblem

The general idea of the solution for this subproblem is similar to the easy subproblem. Again, we will calculate the optimal strategy and its expected profit backwards. However, the computations are a little bit more involved.

The easy subproblem is a stateless game – you don't care about your past, only your future matters. In the hard subproblem your past matters as well – for instance, the cards you threw away influence the expected value of your next draw.

The simplest way to define the state of the game is to define it as the *sequence* of cards you threw away. Of course, there would be over 52! different states, which is way too much. A better attempt is to consider *sets* of cards – you don't care about the order in which you threw the cards away, all that matters is whether it is still present in the deck or not. This reduces the number of states to 2^{52} – still way too much, but better.

The next optimization is to note that we don't care whether we discarded the king of hearts or the king of spades. All that matters is the *number of kings* in the deck. So to describe the state of the game, we have to specify 13 integers, each between 0 and 4: for each value, the number of cards with that value that have been discarded. This gives us 5^{13} states, which is about 10^9 – which starts being quite reasonable.

The last small optimization: in the optimal solution you clearly never discard a king – you cannot get more than 13 points. So we just need to focus on the 5^{12} (approx. 244 million) states in which all kings are still in the deck. The states can be encoded into integers from 0 to $5^{12} - 1$.

Before we move on to the next part of the solution, one final note on defining the state. It may be tempting to include the card you are holding in your hand into the state. There are valid solutions

that define the state in this way, but their number of states is larger. (To see that both approaches are correct, just note that one considers the states as moments immediately *before* and the other as moments immediately *after* you draw a card from the deck.)

For each of the states, as defined above, we want to answer the question: "What is my expected profit, if I start in this state and play the rest of the game optimally?" The answer for state 0 (state in which no cards were discarded yet) is then the answer to our original problem.

(Note that the answer to the question depends not just on the state, but also on the number of turns left to play. For each x we have to do the whole computation from scratch.)

How does the computation for a particular state s look like? From s we know the current contents of the deck. Each value v has some probability of being the next one drawn. And each v determines a new state s' – the state reached from s by discarding a card with value v. If we are doing the computation backwards, we already know the expected profit from the state s', hence we know whether in state s we want to keep or discard the card with value v. The expected profit for s is then computed as a weighted average of expected profits for all possible v.

The algorithm can be implemented using a single huge array of doubles. The indices to the array correspond to states in the game, the values represent expected profits. The values in the array can be computed backwards from the largest value to the value 0. When a state is processed, it is decoded into a representation of already drawn cards, the considered drawn card is added to the description to build the next state, this state is encoded into a number and that is used to index into the array. Note that an addition of a card always produces a larger index than the current one, hence the new index always points into the already computed part of the array.

Finally, note that if we handle the kings separately, we create a special case that has to be handled in the implementation: whenever $x > 48$, we are actually certain to get a king if we throw away enough cards, so the expected profit is exactly 13.

Problem D: Data mania

This year, we, the IPSC organizers, finally decided to rewrite our old webpage and grading system. The old code was full of hacks and it looked more like spagetti code than anything else. During this conversion, we found out that we have quite a lot of data about previous years (teams, submissions, etc.). One of the things that is still missing (as you can see on our page) is the hall of fame. However, we would like to make it less boring than just "winners of the year XY". Thus, we need to analyze our data and calculate the most crazy statistics like "The names of teams with the longest accepted submissions". But as you can guess, we are now quite busy with the preparation of the contest – the new grader is being finished as we speak, and we still need to finish some tasks, including this one. Therefore, you should help us!

Problem specification

You will be given a set of five data tables about previous three IPSC contests (2009-2011). We already did a good job in cleaning these data so you can assume that it is consistent (we will explain later what this means). Note that these data file *are common for both subproblems* (easy and hard).

Your task is to compute results of several data analytics queries that are given in plain English.

Data files specification

The supplied data will consist of the following tables: problems, teams, team members, submissions and submission results. The files are named "`d-problems.txt`", "`d-teams.txt`", "`d-members.txt`", "`d-submissions.txt`" and "`d-messages.txt`".

In general, each table consists of several space-separated columns. Each column holds a string consisting of one or more permitted characters. The permitted characters are letters (`a-z`, `A-Z`), digits (`0-9`), underscores, dashes and colons (`_-:`). All other characters (including spaces) were converted to underscores.

The file `d-problems.txt` contains three space-separated columns. The first column contains the year of the contest, the second column contains the task (one letter, A-Z), and the third column contains the (co-)author of the problem. If the task had multiple co-authors, they will be listed on separate lines.

The file `d-teams.txt` contains five space-separated columns. The first column contains a TIS (a unique identifier of a team; also, teams in different years of the contest never share the same TIS). The second column contains the name of a team, the third column contains its country, the fourth column contains the name of its institution, and the fifth column contains the division ("open" or "secondary").

The file `d-members.txt` contains three space-separated columns. The first column contains the TIS of a team, the second column contains the name of one team member, and the third column contains the age of that member (or zero if age was not specified). If the team had more than one team member, each team member will be listed on a separate line.

The file `d-submissions.txt` contains six space-separated columns. Each row corresponds to one submission received by our server. The first column contains the date when the submission was received in the format "YY-MM-DD". The second column contains the corresponding time in the format "hh:mm:ss". The third column is the TIS of the team who sent the submission. The fourth column is the subproblem identifier. It consists of a letter and a number (1 for easy, 2 for hard), such as "A2" or "G1". The fifth column contains the size of the submitted file (in bytes). Finally, the last column contains the hash of the submitted file.

The file `d-messages.txt` contains four space-separated columns. Each row describes a message shown to one of the teams. The first column contains the TIS of the team. The second column contains a unix timestamp of the moment when the message was created. The third column contains subproblem identifier (in the same format as above). Finally, the fourth column contains the status message which is either

"OK" or "WA". (Note that during actual contests there were several other possible messages including "contest is not running" and "too many submissions". Those were removed when we cleaned up the data.)

You may assume the following:

- For the purpose of conversion between unix timestamps and dates our server runs in the timezone +02:00.
- Each contest starts exactly at noon and lasts until 16:59:59. (Note: we shifted the 2009 contest by -2 hours to guarantee this.)
- The triples (TIS, timestamp, subproblem) are unique, i.e., there are no two concurrent submissions of the same problem from the same team.
- Submissions and messages can be paired using the triple (timestamp, TIS, subproblem). That is:
  - For each submission, there is exactly one corresponding message with its evaluation.
  - For each message there is exactly one corresponding submission.
- Submissions and messages are consistent with problems.
  - For each submission/message, the subproblem is a valid subproblem for that year, i.e. there is a corresponding problem for that year.
  - For each year and the problem, there are exactly two subproblems (easy and hard) and both contain at least one submission.
- For each submission there is a team with the corresponding TIS.
- For each year and subproblem, each team has at most one submission that received the message "OK". That submission, if it exists, is the last one in chronological order.
- For each team there is at least one member.
- For each team there is at most three members.
- For each team member there is a team with the corresponding TIS.
- The country name and institution name may be arbitrary strings. They do not have to represent actual countries and institutions. Note that they are also case sensitive.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of one or several consecutive lines describing the analytics query in plain English. Unless otherwise stated, you should assume that

- "subproblem" means year + subproblem,
- a team is identified by its TIS, not by its name,
- an accepted submission is a submission that received the message "OK".

Output specification

For each testcase, output several lines. On the first line, output an integer c – the number of results for that testcase. The next c lines should contain the results as tuples, with elements of a tuple separated by single spaces. (Note that the format of the output is very similar to the format of the data files.)

If the results should be sorted according to some sorting criteria, proceed as follows: Integers and doubles should be sorted according to their numerical value. Strings should be sorted lexicographically according to their ASCII value. Years should be sorted chronologically (in increasing value).

Moreover, if the query involves string comparison, the comparison should be done using ASCII values. You may assume that no query will require comparison of floating-point values for equality.

Example

input

```
2

For each year (in chronological order)
find the first accepted submission.
Print the year, the name of the team
that made it and the time from the start
of the contest. Time should be formatted
as "mm:ss".

For each year (in chronological
order) print the year, the number of
OK submissions and the number of WA
submissions in that year.
```

output

```
3
2009 Bananas_in_Pyjamas 02:54
2010 Gennady_Korotkevich 02:28
2011 Never_Retired 06:16
3
2009 3352 2942
2010 4441 5579
2011 2437 3340
```


Task authors

Problemsetter:	Vlado 'Usamec' Boža
Task preparation:	Peter 'PPershing' Perešíni, Vlado 'Usamec' Boža

Solution

In this problem, you were given several *tables* from the IPSC *database* and you were supposed to perform several data analytics *queries* which computes *counts, averages* of data *aggregated* by several *columns*. Finally, you had to *order* the results. Does this sound familiar? Well, yes, it seems like SQL.

So, in a nutshell, there are two possible solutions for this task.

No-SQL

In a solution that does not use SQL you need to make all the queries by yourself. At first, this seems to be a lot of trouble. However, most of the testcases from the easy subtask just required counting and sorting. For the hard subproblem, the easiest way was to merge submissions and messages into one data structure. This can be easily done, for example by using hashmap addressed by the unique key (TIS + timestamp + subproblem). Generally, we would recommend also creating hashmaps for teams, members and submit-messages addressed by TIS.

Once you hava the data in this format, it is relatively straighforward to iterate over the desired table or join of tables.

SQL

When we first created this task, we thought that SQL will be a silver bullet that will solve all your problems. After spending one or two sleepless nights over it, we still think that it is a silver bullet. However, you absolutely have to know where you aim – otherwise you may shoot yourself! Why is that? Well, SQL is a good slave but you need to know several of its quirks. Here we will present the most interesting ones:

Consistent data: Needless to say, after extracting our data from different sources (database, old logs) and applying some adjustements and filterings (shifting times in 2009, removing messages different from OK/WA, removing concurrent submissions by the same team[1]) our dataset was a big emmental cheese. Making it consistent (and especially figuring out what "consistent" means) was not trivial. Nevertheless, you were spared of this tedious task.

Timezone conversions: For converting between DATETIMEs and unix timestamps (the function FROM_UNIXTIME), you need to have a correct timezone. In MySQL, you need to run this query first: `SET SESSION time_zone = "+02:00"`

String comparisons: Didn't we stress it enough in the problem statement that the strings should be compared case-sensitive? What a surprise if you try following query `SELECT 'x'<'X', 'x'='X', 'x'>'X'` in MySQL and receive false, true, and false. The correct solution to this problem is to use the setting `CHARSET ascii COLLATE ascii_bin` for all your tables.

However, this still does not solve the special quirk of MySQL which is trailing space removal. Just try (even with binary collation) `SELECT 'x' = 'x  '`. The solution to this second problem is the keyword

[1] Honestly, we do not know how some team managed to submit the same task twice in the same second

BINARY. Fortunately for you, there was no way to smuggle trailing whitespaces inside the dataset (although we were really considering just changing the separator to semicolon when we learned about this "feature").

JOIN, LEFT JOIN and WHERE: In short, consider the testcase "For each year and subproblem, print the count of accepted submissions". A naive solution looks like this:
`SELECT year, subproblem, COUNT(*) FROM messages WHERE message='OK'`
but you may see that the results will not contain the subproblems that were never solved.

One possible fix is this:

```
SELECT year, subproblem, COUNT(IF(message='OK','anything', NULL)) FROM messages
```

Another possibility is:

```
SELECT T1.year, T1.subproblem, T2.cnt FROM
(SELECT DISTINCT year, subproblems FROM messages) as T1
LEFT JOIN
(SELECT year, subproblem, COUNT(*) FROM messages WHERE message='OK') T2
ON T1.year = T2.year and T1.subproblem = T2.subproblem
GROUP BY T1.year, T1.subproblem
```

LEFT JOIN and COUNT: Guess which of these queries works for the above test case:

```
SELECT T1.year, T1.subproblem, COUNT(*) FROM messages
LEFT JOIN
(SELECT year, subproblem, COUNT(*) FROM messages WHERE message='OK') T2
ON T1.year = T2.year and T1.subproblem = T2.subproblem
GROUP BY T1.year, T1.subproblem
```

versus

```
SELECT T1.year, T1.subproblem, COUNT(message) FROM messages
LEFT JOIN
(SELECT year, subproblem, COUNT(*) FROM messages WHERE message='OK') T2
ON T1.year = T2.year and T1.subproblem = T2.subproblem
GROUP BY T1.year, T1.subproblem
```

Hint: COUNT(*) counts the number of rows, COUNT(message) counts the number of non-null messages. Consider cases where LEFT JOIN generates NULL in message column.

To wrap this up: This task can be easily solved with SQL but some level of SQL-fu is necessary. There are a lot of bugs lurking around in the seemingly simple SQL queries.

Bonus: It turned out that even our extensive testing did not catch all problems. The problem we did not foresee was a rounding problem while calculating average penalty for year 2009, subproblem F1 (see hard input). The exact value "212.125" should be rounded to two decimal places and different programming languages not always agree on the final value. In fact, SQL itself is inconsistent (even though the sum of penalties is an integer, and the value 0.125 has a finite expansion in base 2). Consider this output from MySQL:


```
+------------------------+-------------+-------------------+
| ROUND(avg(penalty), 2) | avg(penalty) | ROUND(212.125, 2) |
+------------------------+-------------+-------------------+
|                 212.12 |      212.125 |            212.13 |
+------------------------+-------------+-------------------+

+-------------------+-------------+----------------+
| ROUND(5091/24, 2) | sum(penalty) | count(penalty) |
+-------------------+-------------+----------------+
|            212.13 |         5091 |             24 |
+-------------------+-------------+----------------+
```


Problem E: Evil matching

Consider two sequences of **positive** integers: $T = t_1, t_2, \ldots, t_n$ (text) and $P = p_1, p_2, \ldots, p_m$ (pattern). The classical definition says that P matches T at position k if $p_1 = t_k$, $p_2 = t_{k+1}$, ..., and $p_m = t_{k+m-1}$. One can easily find all positions where P matches T – for example, by using the Knuth-Morris-Pratt algorithm.

Easy is boring. Let's make it a bit harder. We say that P *evil-matches* T at position k if there is a sequence of indices $k = a_0 < a_1 < \cdots < a_m$ such that:

$$\begin{aligned} t_{a_0} + t_{a_0+1} + \cdots + t_{a_1-1} &= p_1 \\ t_{a_1} + t_{a_1+1} + \cdots + t_{a_2-1} &= p_2 \\ &\cdots \\ t_{a_{m-1}} + t_{a_{m-1}+1} + \cdots + t_{a_m-1} &= p_m \end{aligned}$$

In other words we are allowed to group and sum up consecutive elements of the text together before matching them against the pattern. For example, if we have $T = 1, 2, 1, 1, 3, 2$ and $P = 3, 2$, then there are two evil-matches: one at $k = 1$ (with $a_1 = 3, a_2 = 5$), the other at $k = 5$ (with $a_1 = 6, a_2 = 7$).

The *evil compatibility* of a text T and a pattern P is the number of different k such that P evil-matches T at position k.

Problem specification

Given a text T and two patterns P_1, P_2 calculate:

1. The evil compatibility of T and P_1.
2. The evil compatibility of T and P_2.
3. The smallest **positive** integer n for which the evil compatibility of T and $P_1 \cdot n \cdot P_2$ is maximized. (The symbol $\cdot$ represents concatenation.)
4. The evil compatibility of T and $P_1 \cdot n \cdot P_2$ for that n.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of six lines:

- The first line contains the number n – the length of text.
- The second line contains n positive integers – the elements of text.
- The third line contains the number m_1 – the length of the first pattern.
- The fourth line contains m_1 positive integers – elements of the first pattern.
- The fifth line contains the number m_2 – the length of the second pattern.
- The sixth line contains m_2 positive integers – elements of the second pattern.

You may assume the following constraints:
In each test case of the easy data set: $n \leq 5000$, $m_1, m_2 \leq 600$, and the sum of all elements of $T+P1+P2$ does not exceed 11 000.
In each test case of the hard data set: $n \leq 11 \cdot 10^6$, $m_1, m_2 \leq 2 \cdot 10^6$, and the sum of all elements of $T + P1 + P2$ does not exceed $22 \cdot 10^6$.

Output specification

For each test case output a single line.
This line should contain four space-separated numbers as described in problem specification.

Example

input

```
1

13
1 1 1 1 1 47 1 1 1 1 1 1 1
3
1 1 2
3
1 1 1
```

output

```
6 8 48 2
```

The first pattern evil-matches the text at positions 1, 2, 7, 8, 9, and 10.
The second pattern evil-matches the text at positions 1, 2, 3, 7, 8, 9, 10, and 11.
The pattern "1,1,2,48,1,1,1" evil-matches the text twice – at position 1 and at position 2.

Note that the value $n = 48$ is indeed optimal:
If $1 \leq n < 47$, the pattern "1,1,2,n,1,1,1" does not evil-match the text at all.
The pattern "1,1,2,47,1,1,1" evil-matches the text only at position 2.
There is clearly no $n > 48$ such that the pattern "1,1,2,n,1,1,1" evil-matches the text at three positions.

Task authors

Problemsetter: Vlado 'Usamec' Boža
Task preparation: Vlado 'Usamec' Boža, Peter 'PPershing' Perešíni

Solution

The easy data set can be solved by brute force. It is easy to check for an evil match of P and T at k in $O(n+m)$ time. To compute the evil compatibility of P and T, just iterate over all k.

There are at least two ways how to answer the last two queries.

You may simply compute the evil compatibility for the text T and each pattern $P_n = P1 + n + P2$ such that the sum of P_n does not exceed the sum of T.

Alternately, you can take all pairs (an evil match of P_1, an evil match of P_2 to the right of the first match) and for each of them compute the corresponding n as the sum of T between the two matches.

We will now convert sequences of positive integers into sequences of zeroes and ones. Define $E(x)$ as a 1 followed by $x-1$ zeroes. E.g. $E(5) = 10000$. Let S be a sequence of positive integers $s_1, s_2, \ldots, s_k$. Then define $E(s) = e(s_1) \cdot e(s_2) \cdot \cdots \cdot e(s_k) \cdot 1$ (where $\cdot$ represents concatenation).

Given this expansion we can find evil matches of P in T using the following idea: P matches T in some position if and only if every 1 in $E(P)$ matches a 1 in $E(T)$. Of course, directly checking this is actually slower than the brute force solution we described above. However, this solution can be improved to be way faster.

For a sequence S of zeroes and ones we define that $\overline{S}$ is the sequence in which we exchange all zeroes for ones and vice versa. Also, let S^R be the reversal of sequence S. For example, if $S = 000110$, then $\overline{S} = 111001$ and $S^R = 011000$.

Now consider the sequences $E(P) = a_0, a_1, \ldots, a_{x-1}$ (pattern) and $\overline{E(T)} = b_0, b_1, \ldots, b_{y-1}$ (negated text). Then each evil match corresponds to some z such that

$$b_z a_0 + b_{z+1} a_1 + \cdots + b_{z+x-1} a_{x-1} = 0$$

Now wait, that starts to resemble the result of polynomial multiplication. For example, if you multiply:

$$(p_3 x^3 + p_2 x^2 + p_1 x + p_0) \cdot (q_2 x^2 + q_1 x + q_0)$$

the coefficient of x^3 is $p_3 q_0 + p_2 q_1 + p_1 q_2$.

Okay, but in our formula the indices in both sequences increase. To fix that, we will simply reverse the pattern. Let $E(P)^R = c_0, c_1, \ldots, c_{x-1}$. Then evil matches between P and T correspond to indices z such that

$$b_z c_{x-1} + b_{z+1} c_{x-2} + \cdots + b_{z+x-1} c_0 = 0$$

So in order to count the evil matches between P and T, we just need to take the polynomials that correspond to $\overline{E(T)}$ and to $E(P)^R$ and multiply them. Once you have their product, you just count the relevant zero coefficients. (The multiplication should be done by some fast algorithm, such as FFT or possibly Karatsuba.)

Now to the last part of the problem. This turns out to be just one more convolution. By solving the first part, for each pattern we can create an array of ones and zeroes, where zero represents a match. We can now reverse one of these two arrays and use a fast polynomial multiplication again to compute the best n.

Problem F: Fair coin toss

Lolek and Bolek are studying to become magicians. For the last hour they have been arguing whose turn it is to take out the trash. The fair solution would be to toss a coin... but one should never trust a magician when tossing a coin, right?

After looking through all pockets, they found n coins, each with one side labeled 1 and the other labeled 0. For each coin i they know the probability p_i that it will show the side labeled 1 when tossed.

Of course, just knowing their coins did not really solve their problem. They still needed a fair way to decide their argument. After a while, Lolek came up with the following suggestion: they will toss all the coins they have, and xor the outcomes they get. In other words, they will just look at the parity of the number of 1s they see. If they get an even number of 1s, it's Bolek's turn to take out the trash, otherwise it's Lolek's turn.

Now they are arguing again – about the fairness of this method.

Problem specification

A set of coins is called *fair* if Lolek's method is fair – that is, the probability that Bolek will take out the trash has to be exactly 50%.

You are given a description of all coins Lolek and Bolek have.

As a solution of the **easy data set F1** find out whether these coins have a fair *subset*.

As a solution of the **hard data set F2** find out *how many* subsets of the given set of coins are fair.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of two lines. The first line contains the number n of coins (at most 10 in the easy data set, at most 60 in the hard data set). The second line contains n decimal numbers: the probabilities $p_1, \ldots, p_n$. Each probability is given to exactly 6 decimal places.

Output specification

For each test case output a single line.

For the easy data set this line should contain "0" if the given set has no fair subset, and any positive integer not exceeding 2^{60} if it contains at least one such subset.

For the hard data set this line should contain the exact number of fair subsets.

(Note that a program that correctly solves the hard data set should also solve the easy data set.)

Example

input

```
2

3
0.500000 0.500000 0.500000

4
0.000001 0.000002 0.000003 0.000004
```

output

```
7
0
```

In the first test case each subset is obviously fair. In the second test case no subset is fair. When tossing a subset of these coins, you are almost certain to see all zeroes – Bolek would be really unhappy!

Task authors

Problemsetter: Michal 'mišof' Forišek
Task preparation: Ján Hozza, Monika Steinová, Michal 'mišof' Forišek

Solution

The moral of this story: If none of your coins are fair, Lolek's attempt never works. (And if you have a fair coin, there is no need for elaborate shenanigans.)

Lemma 1. Assume that Lolek and Bolek decided to toss k coins. If at least one of the k coins is fair, the outcome of their game is fair. (That is, each of them loses with probability 50%.)

Proof. Just imagine that we toss the coins one at a time, and the last coin we toss is fair. Before tossing the last coin, one of the boys is currently winning and the other one is currently losing. The last coin will change this with probability exactly 50%.

The same, formally. Pick an order in which the coins are tossed. Let p be the probability that the bitwise xor of the outcomes of first $k-1$ tosses is 1. Then the probability of the final outcome being 1 is $0.5 \cdot p + 0.5 \cdot (1-p) = 0.5$. □

Lemma 2. Assume that Lolek and Bolek decided to toss k coins. If none of the k coins is fair, the outcome of their game is not fair.

Proof. We will use induction on the variable k. The case $k = 1$ is trivial. Now assume that the statement holds for any set of k biased coins. Take any set of $k+1$ biased coins. Tossing these $k+1$ coins is equivalent to first tossing k of them and then the remaining one. From the induction hypothesis we know that after tossing k biased coins, the probability of the outcome 1 is $p \neq 0.5$. For the last coin, the probability of getting a 1 is $q \neq 0.5$.

The final outcome 1 can be obtained in two mutually exclusive ways: either the first k coins xor to 1 and the last coin gives us a 0, or the first k coins xor to 0, and the last coin shows a 1. Hence we can compute the probability of getting the final outcome 1 as $p(1-q) + (1-p)q$.

Now it's just simple algebra. To simplify the formula, let $p = 0.5 + a$ and $q = 0.5 + b$, where $a, b \neq 0$. We can then write:

$$p(1-q) + (1-p)q \;=\; (0.5+a)(0.5-b) + (0.5-a)(0.5+b) \;=\; 0.5 - 2ab \;\neq\; 0.5$$

□

The solution to the competition task now becomes very simple. For the easy data set, we just need to check whether any of the probabilities equals `0.500000`.

For the hard data set, assume that there are f fair coins and $n-f$ biased coins. How do the fair subsets look like? We may take any subset of the biased coins, and any non-empty subset of the fair coins. Therefore the answer is $2^{n-f}\left(2^f - 1\right)$.

Problem G: Gems in the maze

Scrooge McDuck has a new plan how to increase his wealth. He found ancient ruins with an extraordinary maze. This maze consists of n chambers. The chambers are numbered 0 through $n-1$. Each chamber contains exactly one gem. Chambers are connected by one-way tunnels. Each chamber v has exactly two outgoing tunnels: one leads to the chamber with number $(a \cdot v^2 + b \cdot v + c) \mod n$, the other will bring you out of the maze.

You can enter the maze at any location, move along the tunnels and collect the gems. But once you leave the maze, you'll trigger a self-destruct mechanism – the ceiling of the maze will collapse and all the gems that you did not collect will be lost forever.

Scrooge wants to know the maximum number of gems he can take from the maze.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consist of a single line containing four integers a, b, c, and n – the numbers that describe one particular maze.

In the easy data set $n \leq 300$. In the hard data set $n \leq 2^{24}$.

Output specification

For each test case, output a single line containing a single integer – the maximum number of gems that can be taken from the maze.

Example

input

```
3

1 2 0 64

0 2 1 47

0 3 5 128
```

output

```
5
23
64
```

The starting chamber matters. For instance, assume that in the first example test case Scrooge starts in the chamber 0. His only two options are a tunnel that leads back to chamber 0 and a tunnel that leads outside – not much of a choice. A much better strategy is to start in the chamber 2 and follow the path $2 \to 8 \to 16 \to 32 \to 0 \to$ *outside.*

Task authors

Problemsetter:	Vlado 'Usamec' Boža
Task preparation:	Michal 'Mic' Nánási, Vlado 'Usamec' Boža

Solution

We can forget about the tunnels that leave the maze and view it as a directed graph with vertices of outdegree 1. If there is an edge $u \to v$ in our graph, we say that v is the successor of u. The successor of u will be denoted $s(u)$. In our graph we are searching for a walk (a sequence of consecutive vertices) that contains the highest number of distinct vertices.

The first observation is that once we visit a vertex for the second time, all the following vertices will be already visited as well (and therefore we will not find any more gems). This is because the edges never change and our movement is deterministic. Once we made a sequence of steps that brought us from some vertex v back to v, we will be repeating the same sequence of steps over and over. Also, as there is only a finite number of vertices, sooner or later a vertex has to repeat in any walk. Therefore all we need to do is to search for the longest simple path (a walk in which all the vertices are distinct).

We now know enough to solve the easy data set: for each possible starting room, we simulate the walk until a vertex repeats, and then output the length of the longest walk constructed in this way.

For the hard data set we need a faster solution. We will show one that is linear in the graph size – i.e., with time complexity $O(n)$.

Let $D(v)$ be the number of gems we will collect if we start at the vertex v.

Let's pick a vertex v and construct the sequence of its successors: $a_0 = v$, $a_1 = s(a_0)$, $a_2 = s(a_1)$, and so on. Let x be the smallest integer such that $a_y = a_x$ for some $y < x$. What can we tell about the values $D(a_\star)$? Actually, we now know all of them precisely. The vertices $a_y, \ldots, a_{x-1}$ lie on a cycle. For each of them the number of gems is the length of the cycle: $D(a_y) = \cdots = D(a_{x-1}) = x - y$. For any previous vertex the number of gems is larger. If we start at some a_i with $i < y$, we collect $y - i$ gems before reaching the cycle, and then $x - y$ gems on the cycle, which gives us $D(a_i) = x - i$.

Now we just need one more improvement: once you calculated D for some vertex, reuse it and never process this vertex again. How does this look like in practice? Suppose we alredy processed the entire sequence for some vertex v, as shown above. Now let's take an unprocessed vertex w and start building its sequence $w = b_0, b_1, b_2, \ldots$. If we happen to reach some x such that we already know $D(b_x)$, we may immediately stop. At this moment we know that the vertices b_0 through b_{x-1} do not lie on a cycle and that their D values can be computed backwards from $D(b_x)$: we have $D(b_{x-i}) = D(b_x) + i$.

To summarize this solution in pseudocode:
For each vertex:
    If this vertex was not processed yet:
        Construct the sequence of successors until a vertex repeats or a processed vertex is reached.
        Going backwards, compute D for each vertex in the sequence.
        Mark all vertices in the sequence as processed.
Output the value $\max_v D(v)$.

Another view of exactly the same solution is that $D(v)$ can be computed using a depth-first search. However, a recursive implementation may have (depending on the environment you use) problems with exceeding the stack limit – some test cases in the hard data set contain very long cycles. The above solution basically describes a depth-first search in which the sequence represents the stack.

Problem H: Harvesting potatoes

There is a rectangular field divided into $r \times c$ square cells. Potatoes grow on all cells of the field. You have a potato harvester and want to harvest all of them. A potato harvester is basically a large car equipped with mechanical spades and other stuff necessary to get the potatoes from the soil to the back of the car. The driver has a switch that turns the mechanical spades on and off.

The harvester operates in passes. In each pass it traverses either a single row or a single column of the field, starting at one border and ending at the opposite one. The capacity of the harvester is limited: in each pass it can only harvest at most d cells. These can be any of the cells it passes through. In this task your goal will be to produce a harvesting schedule that uses *the smallest number of passes*.

In practice there is one more factor: harvester drivers are bad at following complicated instructions. Harvesting contiguous segments is much simpler than repeatedly turning the spades off and on again at the right places. The *difficulty of a given pass* can be measured as the number of times the driver has to flip the switch during the pass. In other words, the difficulty is twice the number of contiguous segments of cells harvested in that pass. (Note: The driver must harvest each cell *exactly* once. It is *not allowed* to leave the spades on while the harvester passes over already harvested cells, as this damages the soil.)

The difficulty of a schedule is the difficulty of the most complicated pass it contains, i.e., the maximum over all difficulties of individual passes.

Problem specification

You will be given the dimensions r and c and the capacity d. For the **easy data set H1** your task is to produce any harvesting schedule with the smallest number of passes. For the **hard data set H2** your task is to produce any harvesting schedule that solves the easy data set and additionally has the smallest possible difficulty (out of all such schedules).

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line. Each test case is given as a single line with three integers: r, c, and d.

Each test case in the easy data set has $r, c \leq 20$. Each test case in the hard data set has $r, c \leq 400$.

Output specification

For each test case, output r rows, each containing c space-separated *positive* integers. These represent the harvesting schedule. More precisely, each integer gives the number of the pass in which the particular cell is harvested. (Additional whitespace is OK as long as the number of tokens in your output is correct.)

Example

input

```
2

2 9 5

3 5 2
```

output

```
1 1 1 1 1 2 2 2 2
3 3 3 3 3 4 4 4 4

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

The first test case: The cells are harvested in rows. The second test case: We first do one pass in each column and then three passes in the last row. The output shows a schedule that is shortest but does not have the smallest possible difficulty – to fix that, swap the 8 with one of the 7s.

Task authors

Problemsetter:	Monika Steinová, Michal 'mišof' Forišek
Task preparation:	Monika Steinová, Michal 'mišof' Forišek

Solution

The entire task can be solved by a careful case analysis. Alternately, there is a solution based on maximum flow (that works at least for the easy subtask). Here is a summary of the case analysis:

- Narrow strips of land (less than d wide) can be harvested optimally by a simple schedule with difficulty 2 (i.e., one that harvests a contiguous strip of cells in each pass).
- Any other field can be harvested in exactly $\lceil rc/d \rceil$ passes, which is clearly optimal.
- Moreover, there always is a shortest schedule with difficulty at most 4.
- Sometimes, but not always, there is also a shortest schedule with difficulty 2.

General observations

While harvesting the field, we can only harvest d cells in each pass, hence we need at least $\lceil rc/d \rceil$ passes. This will usually be possible, except for some very narrow test cases. (Note that once you have any schedule with $\lceil rc/d \rceil$ passes, you can be sure that it is shortest.)

The difficulty of a pass is always even – the harvester has to be switched on and then off. Hence the minimum difficulty of a schedule is 2 and the second best we can get is 4. We will later show that there is always some shortest schedule with difficulty at most 4 – in each pass at most two contiguous segments are harvested. The core of the problem lies in discovering such a schedule, and in determining when there is also a shortest schedule with difficulty 2.

Solving narrow fields

First we consider test cases where d exceeds one of the dimensions. W.l.o.g. we will assume that $r < d$.

Consider any schedule. Let x be the number of vertical passes. What is the smallest number y of additional horizontal passes we need in order to harvest everything?

Each of the vertical passes may as well harvest the entire column – there is no point in skipping some of the cells. Now consider any row. Originally we had to harvest c cells in this row, we already harvested x, hence we need at least $\lceil (c-x)/d \rceil$ horizontal passes in this row. We can conclude that if we make x vertical passes, the total number of passes will be at least $f(x) = x + r\lceil (c-x)/d \rceil$.

To find the smallest number of passes, we will now find the x for which $f(x)$ is smallest, and then show a schedule that actually uses that many passes. Without much thinking, we can simply try all possible x, pick the one with the smallest total number of passes, harvest the first x columns, and then harvest the rest of each row from the left to the right.

(Alternately, we can actually compute the optimal x easily. It is either 0 (if $c \bmod d > r$) or $c \bmod d$ (otherwise). To see this, first observe that the optimal x is less than d, because increasing x by d increases $f(x)$ by $d - r > 0$. For $x < d$ the value $x = c \bmod d$ is the only one where f locally decreases, so there are no other candidates for the minimum.)

Note that the above algorithm not only produces a shortest schedule, we can also guarantee that the difficulty of this schedule will be 2.

Solving other fields with difficulty 4

By far the most common mistake was not finding a schedule with $\lceil rc/d \rceil$ passes in some tricky cases. One of the smallest examples is $r = c = 6$, $d = 4$. Most teams who attempted this task only found a solution that needs 10 passes. However, $(6 \cdot 6)/4 = 9$, so theoretically there can be a better solution – and in fact, it does exist. Example:

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

There were only a few such test cases in the easy data set. If you managed to identify them as a possible source of getting all the "Wrong answer"s, one way of dealing with them was to solve them on paper and to hard-wire the solution.

We will now show a general solution: we will prove that whenever $r, c \geq d$, there is always a schedule with exactly $\lceil rc/d \rceil$ passes (which is clearly shortest), and difficulty (at most) 4.

First, we reduce the problem to a situation where $r, c < 2d$. This is easily done by horizontal/vertical passes that harvest contiguous strips of length d.

Now, let $r = d + y$ and $c = d + x$. The total area is $d^2 + dy + dx + xy$, hence we need $d + x + y + \lceil xy/d \rceil$ passes. We will make $d + y = r$ horizontal passes (one in each row) and $v = x + \lceil xy/d \rceil$ vertical passes (one in each of the first v columns).

One possible schedule looks as follows: In column i (for $0 \leq i < v$) the pass will harvest d contiguous cells, starting at $\lfloor ir/v \rfloor$ and wrapping around if necessary. Then, in each row the pass will harvest the remaining cells.

Example: $r = 7$, $c = 6$, $d = 4$. We have to make 11 passes = 7 rows and the first 4 columns. The pattern described above looks as follows:

```
*..*..
**.*..
**....
***...
.**...
..**..
..**..
```

(Stars are the cells harvested in vertical passes, dots will be harvested in horizontal passes.)

Clearly this schedule has difficulty at most 4. To prove this, just note that if we view the left v columns as a torus, each row segment of dots and each column segment of stars is contiguous. We include a rigorous proof that this solution always works at the end of this problem statement. (For a rough idea, note that our formula "spreads the stars evenly", so all rows get the same number of stars, plus minus one.)

A lower bound for difficulty 2

Notice the following subset of cells: $Z = \{(i, j) \mid i = j \pmod d\}$. (This is every d-th diagonal of the field, including the one starting at $(0, 0)$ and passing through $(1, 1)$.)

Each single contiguous pass will harvest at most one of these cells, because every two cells of Z that share a column or a row are more than d cells apart. Hence for any schedule with difficulty 2 the number of passes must be at least $|Z|$ (i.e., the number of elements in set Z).

Given a $r \times c$ rectangle, $|Z|$ can be computed as follows: any $d \times x$ rectangle contains exactly x elements of $|Z|$. By removing two such rectangles we can easily reduce the $r \times c$ rectangle to a $(r \bmod d) \times (c \bmod d)$ rectangle containing $(0, 0)$, and for this rectangle obviously $|Z| = \min(r \bmod d, c \bmod d)$.

The computation above can easily be turned into a harvesting schedule that uses $|Z|$ contiguous passes: In each row harvest horizontal contiguous segments of d cells, always going from the rightmost non-harvested column to the left. This will leave you with a $r \times (c \bmod d)$ rectangle. In each of the columns harvest vertical contiguous segments of d cells starting at the bottom. This leaves a rectangle of $(r \bmod d) \times (c \bmod d)$ cells. Harvest this remaining rectangle by horizontal passes (if $r \bmod d < c \bmod d$) or vertical passes (otherwise).

Whenever $|Z| = \lceil rc/d \rceil$, we can use the above algorithm to generate a fastest schedule with difficulty 2. Whenever $|Z| > \lceil rc/d \rceil$, we can be sure that there is no fastest schedule with difficulty 2, hence we can use the schedule with difficulty 4 we showed in the previous section.

A proof of the difficulty-4 solution

We need to show that after the column passes each row contains at most d unharvested cells ("dots").

The following proof only looks at the left v columns and treats them as a torus – i.e., all indices are considered modulo the respective sizes.

Pick any i. Consider all rows that were harvested in column $i-1$ and are not harvested in column i. In the next few columns these rows will not be harvested. Clearly it is enough to focus on the last of these rows. This is row $p = \lfloor ir/v \rfloor - 1$.

When will row p be harvested again? In column j, where $j > i$ is the smallest index such that $\lfloor jr/v \rfloor - \lfloor ir/v \rfloor \geq y$. If we find j, we may conclude that the number of dots in row p is $(j - i)$ plus the dots in the $c - v = d - \lceil rc/d \rceil$ columns that are not harvested vertically.

Lemma 1. Let $j = i + \lceil xy/d \rceil$. Then $jr - ir \geq yv$.

Proof. Let $q = \lceil xy/d \rceil$. Note that $qd \geq xy$. Applying the assumption we have

$$jr - ir = (j - i)r = q(d + y) = qd + qy \geq xy + qy = y(x + q) = yv.$$

□

Theorem 1. Let $j = i + \lceil xy/d \rceil$. Then $\lfloor jr/v \rfloor - \lfloor ir/v \rfloor \geq y$.

Proof. From the lemma it follows that $jr/v \geq y + ir/v$, the statement to prove follows easily. □

From the theorem it follows that j cannot be larger than $i + \lceil xy/d \rceil$. To conclude our reasoning we compute that the overall number of dots in row p is at most

$$(j - i) + (c - v) = (i + \lceil xy/d \rceil - i) + (d - \lceil rc/d \rceil) = d + \lceil xy/d \rceil - \lceil rc/d \rceil \leq d$$

Hence there will be no more than d dots in any row of the entire field.

An alternate solution: maximum flow

We will just sketch the general idea of this solution. First, we can show that any large test case can be reduced to one where $r, c < 2d$ – it never hurts to shave off a strip of width d off a larger test case. Now we can prove (or take a guess and verify by actually running our code) that any such instance can be solved without passing through any row or column twice.

Now we are looking for the smallest x such that the field can be harvested in x passes. We can either increment x until we find one that works, or use binary search to get a more efficient solution. Or we can start at $x = \lceil rc/d \rceil$ and then it actually does not matter much.

Once we fix x, we need to decide how many of the x passes will be horizontal (say y) and how many vertical. Once we fix y, observe that it does not matter which rows and which columns of the field are chosen to be harvested. Hence, w.l.o.g., assume that the first y rows and the first $x - y$ columns of the field are harvested.

We can now build a flow network that models all possible ways of harvesting. There will be two different types of nodes in the network: one node for each of the x passes, and one node for each of the rc cells of the field. The source is connected by an edge of capacity d to each pass-node. A pass-node is connected to every cell-node in the particular row/column that is harvested in that particular pass. Each of these edges has capacity 1. Finally, there is an edge with capacity 1 from each cell-node to the sink. Clearly, a corresponding harvesting schedule exists if and only if the maximum flow through this network has size exactly rc.

Problem I: Invert the you-know-what

Good news everyone! We just "recovered" a password file from the webserver of our arch-enemies!

Input specification

The same input file is used for both data sets. The strings on the left seem to be usernames. The ones on the right... we have no idea. Do you?

Output specification

For each data set you have to submit a file with the following content: a username, a newline, the password for that username, and another newline. (To accommodate multiple operating systems, newlines can be either LF, or CR+LF.)

We will try logging in to the arch-enemies' webserver using that username and password. If the login process succeeds, we will accept your submission.

For the easy data set you may use any valid username. For the hard data set you must use the username `robot7`.

Task authors

Problemsetter:	Michal ‘mišof’ Forišek
Task preparation:	Michal ‘mišof’ Forišek, Vlado ‘Usamec’ Boža

Solution

So, what do we have here? As we were told that this is a password file, we are probably looking at some hashsums. Exactly 32 hexadecimal digits in each of the strings, in other words, $32 \times 4 = 128$ bits. That may be the most common hashsum around: MD5.

So let's try reversing it. The best tool to reverse MD5? Google, of course! If it is something common, it's somewhere on the web already. Of course, there is a bunch of pages that specialize in reversing hashsums, but Google indexes many of them anyway. So just pop a few of the strings into Google. Just like this: `http://www.google.com/search?q=4dba921131c2cec4d2d125a14b954dc5`. Lo and behold, some of them actually do show up in the search results. Here's what you were most likely to find:

- `4dba921131c2cec4d2d125a14b954dc5` is `saltPeter`
- `0b5b2e40acf67fb978de1cd4fbae6461` is `saltb0x3s`
- `533acafff35d56aa156ec9ba9b500b40` is `salt12345`
- `b8b7fbd261d9c405a4ec33281506ca11` is `saltdog`

Now, there's something strange going on here. It's not likely that all of those users would pick passwords starting with `salt`. The most likely explanation is that the person implementing the password check at our arch-enemies' website horribly misunderstood the idea of salting the passwords[2] before hashing them. (If you are still at school, you probably wouldn't believe how much crazy stuff like this goes on in “production code” worldwide.) So the theory here is that the stored strings are computed as `MD5('salt' + password)`. The easiest way to verify this theory is to submit a solution to the easy data set. (If you missed the common prefix and submitted a login:password pair such as `amanda:saltPeter` instead of the correct `amanda:Peter`, you got a slightly helpful message in addition to a wrong answer verdict.)

An alternate solution to the easy data set is to guess one of the passwords, which is doable if you are lucky. For instance, the password of `kbrow` is one row of the keyboard, and the password for `rmunroe` (presumably Randall Munroe?) is the password `correct horse battery staple` of XKCD fame. The two users `dewey` and `louie` apparently use the same password (their hashes are exactly the same). The password turns out to be `huey`. You may also be delighted by knowing that the password for `magritte` was `ceci n'est pas un mot de passe`.

Now that we have solved the easy data set, we need to find the passwords for all the robots. For that we either need to be lucky (and to correctly guess an approximate shape of the robots' passwords) or to roll out the big guns – a password cracker. One option is the notorious John the Ripper: `http://www.openwall.com/john/`. The version 1.7.9-jumbo-5 can handle raw MD5 hashsums.

The fastest way of getting good results is a dictionary attack: take any big wordlist, prepend `'salt'` to each of its lines, and tell John to use this new wordlist:

```
john -wordlist:salt-words -rules -format:raw-md5 passwordfile.
```

If you have a really good wordlist, you are done. With the standard file `/usr/share/dict/words` distributed in Ubuntu (or any comparable wordlist) you should get most of the following passwords:

[2] Using a constant string actually makes sense in some cryptographic protocols, e.g., HMACs. However, for storing passwords random salts are necessary. These then both prevent dictionary-based attacks and have the nice side effect that you will not find out that two users chose the same password.

`robot1:took1`, `robot3:sword3`, `robot4:hand4`, `robot5:long5`, `robot6:time6`, `robot8:sought8`, and `robot9:rested9`. The pattern is clear: the password for `robotX` has the shape `wordX`.

But what's the word for robot 7? It's possible that the words are not random. If a person needed to generate (and remember) passwords for 15 robots, it is quite likely that they would use some easy-to-remember pattern, a phrase or something similar. And if that is the case, again, Google is the best way of finding that pattern.

Once you search for some large enough subset of the robot passwords, the top result will almost surely be the correct one: Lewis Carroll's poem Jabberwocky. "He took his vorpal sword in hand, long time the manxome foe he sought, ..." goes one stanza. The words are clearly in the correct order, and `manxome7` indeed works as the password for `robot7`.

A final note: Sadly, some of the teams were not careful enough and during the contest the hash and the password became available in some of the online databases. Also, at least one team posted the answer as a public code snippet on PasteBin, and Google was quick to index it. So the best strategy for this task turned out to be: if you haven't solved it yet, google for the answer again a few minutes before the end of the contest.)

We would like to encourage all teams to think about the consequences of their actions – in a competition you usually don't want to help the other teams.

Problem J: Jukebox

The IPSC organisers recently installed a new jukebox in their room. As expected, the jukebox was an instant hit. In fact, there have already been several violent exchanges between people trying to select different songs at the same time, but that is a story for another time. This story is about the jukebox and Milan. As a pianist, Milan likes music and especially likes to play the music he hears. Thus it become unevitable that Milan wants to play music from the jukebox. But here is the problem – the jukebox just contains a lot of recorded music and no music sheets.

In order to recover music sheets from the jukebox, Milan already copied all the music to his laptop. Now he just needs to convert these recordings into music sheets. And that, as you surely guessed, will be your task. Fortunately for you, Milan already made some progress, so you will only need to recover the pitches of separate tones.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case describes one raw recording. It starts with a line containing a single integer n – the number of recorded samples from a mono microphone. The next n lines will contain a single integer v_i – numerical value of i-th sample. The sample rate is 22050 Hz.

Moreover, for your convenience we provide MP3 files named `j.{example,easy,hard}.testcase#.mp3` containing the same recordings (except for minor differences due to a lossy MP3 compression).

Output specification

For each test case, output all tones of the song. On the first line, output the count c of tones. On the next lines, output c whitespace-separated tones, where each tone is one of the following strings "`A`", "`A#`", "`B`", "`C`", "`C#`", "`D`", "`D#`", "`E`", "`F`", "`F#`", "`G`", "`G#`".

You may assume that each tone in the input matches one of the twelve musical notes. Note that you do not need to determine the exact octave of the tone.

Example

input

```
2

261820
-533
-863
... (216818 lines)

1535092
-51
... (1535091 lines)
```

output

```
8
C D E F G A B C

84
D A A G C A A G F D ...(74 more tones)
```

The first input is the C major scale. The second input is a traditional Slovak folk song "Tota Helpa" (for some strange reason also known in Japan as Hasayaki).

Task authors

Problemsetter:	Peter 'PPershing' Perešíni
Task preparation:	Peter 'PPershing' Perešíni, Milan 'Miňo' Plžík, Tomi Belan

Solution

The best way to start solving this task is to analyze the stuff we gave you. The mp3 files are very handy for this. Listen to them. On one hand, the problem seems scary – this is clearly a live recording. But on the other hand, it could have been worse. The recordings have two nice properties. They are monophonic (i.e., at most one tone is played at a time) and they are relatively slow, so that there is enough time to recognize each tone without getting consecutive tones mixed up. (Note that we do not really need to detect the lengths of tones.)

An easy solution for the easy dataset

So, how to solve it? There are three possible solutions: The first one is that you may have a team member with absolute pitch. In that case he or she just needs to sit down and transcribe the recordings. However, not many teams will have this luxury.

Another possible approach is to use a program that can convert a wave file to MIDI, and to tweak the output by hand – and ear. Again, this probably works only if you have an ear for music.

For the rest of us, another easy solution is to transcribe the song from some music sheet. Namely, after hearing 3 songs from the easy dataset, one can recognize a chromatic scale consisting of all half-tones (if not sure, just compare to the one from the example); Silent Night and the Tetris theme song.

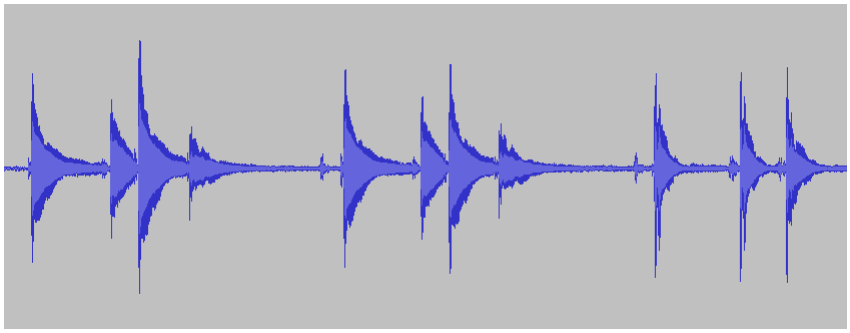
The best way to proceed now is to find a video on YouTube which matches the song and contains its score. For Silent night one such video is `http://www.youtube.com/watch?v=skRJO5J79hs` and for Tetris `http://www.youtube.com/watch?v=mfJ88PAuH24`. When searching for the musical score, you should verify the music you found matches our performance exactly. For example, Silent night can be played starting from a different note. Or there might be some special transitions. (There is a small difference in the Tetris song, check it out.) After acquiring a good source of musical notes, we just need to transcribe them. If you are not musicians, search engines can easily help you finding note-to-name mappings.

Although this strategy is easy, there is one problem. It does not work for the hard testcase. The reason is that there are 3 songs with random notes there. Plus, in Fur Elise, there is one accidental error, which is rather hard to spot. Try our challenge and spot the wrong tone.

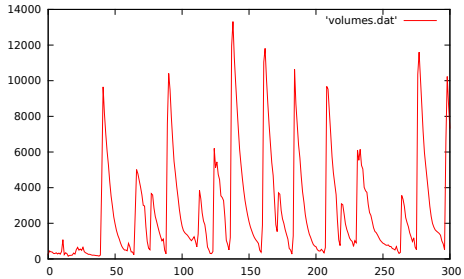
Going the hard way

Everything seems that in the end, you really need to detect tones. The process is two-step. In the first step, one needs to locate (at least approximate) starts of tones. After locating the tones, we may proceed with the recognition itself.

Locating tones: If you open the mp3 files in an audio editor, you may see something like this:



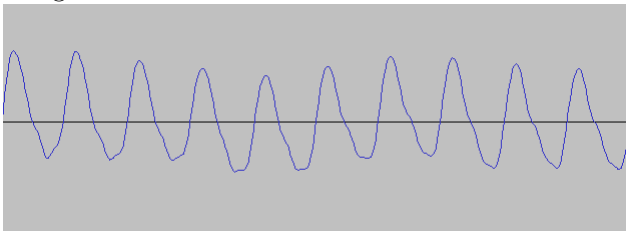
The takeback from the image is that locating tone beginnings should be easy – we just need to create a volume profile of the song and then pick up positions where volume suddenly jumps. One solution is to divide the song into small segments (we recommend size of roughly 0.05s, less is going to be noisy, more is going to blend sharp increases into smaller ones). For each segment calculate the average absolute value of samples and apply some simple filtering technique to detect sudden increases. The volume of blocks might look like this:



and the graph might be handy for estimating "magic" thresholds.

An alternative solution is to simply create a program which records time of your keypresses, play the song and press keys into the rhythm.

Detecting pitch: The first thing that we need to know is how does a tone look like. If we zoom into the sequence, we may see a signal like this:



The tone is (roughly) a periodic signal. The pitch of the tone corresponds to the period, or in other words to the frequency of the signal. Thus, the whole problem is how to detect the frequency of the signal.

If you know at least something from signal processing, you should instantly associate this with Fast Fourier Transform (FFT). FFT takes a sequence of real values as its inputs and produces a sequence of complex values. We care about the absolute value (magnitude) of $result_i$ which corresponds to the amplitude of a sine wave with frequency "i-times per sequence". (The special value $result_0$ just holds the average and it is called DC offset.) If we denote size of the sequence for FFT as $window$ and sample rate as $rate$, then $result_i$ corresponds to the frequency

$$freq = \frac{rate}{window} \cdot i \tag{1}$$

For a while, let us skip the details of how to implement FFT and just assume that we have an efficient algorithm.

One other thing we need to know is the mapping between frequencies and musical notes. Here are the basics:

- The frequency of the reference tone A4 is 440 Hz.
- A shift by an octave up/down corresponds to multiplying/dividing the frequency by 2. For example, A3 is 220Hz and A5 is 880 Hz.
- The 12 half-tones of an octave are uniformly distributed on a logarithmic scale. In other words, the step of a single half-tone corresponds to multiplication/division by $\sqrt[12]{2}$. For example, the frequency of C4, which is three half-tones above A3, is $220 \cdot (\sqrt[12]{2})^3 = 261.63$ Hz.

Knowing that, we can precompute the frequencies of all common musical notes (say from 220 Hz to 1500 Hz). For each of these frequencies we then compute the corresponding i of the FFT result and then pick the one with the biggest magnitude.

There are three technical details here. First, we may misjudge the frequency by an octave. This happens because the same tone (unless it is a pure sine wave) contains also "harmonics". However, this is not a problem because we do not need to output the absolute pitch of the tone, just the tone name (and thus the octave does not matter). The second technicality is the choice of *window*. It cannot be too long because then we might be analyzing two tones. On the other hand, a *window* that is short means low resolution. For example, consider $window = 512$. According to equation 1, for tones A4 and A#4 (frequencies 440 Hz and 466 Hz) we obtain $i_1 = 10.21$ and $i_2 = 10.82$. As i is an integer, we would not be able to differentiate between these two frequencies. Therefore, the length of the window should be at least 2048. Third, the FFT works on periodic data but our sample is not periodic (with the period equal to the length of the data). This might cause an effect called "ringing". There are sophisticated solutions to solve this problem, here we just note that for long enough windows this is not a problem for pitch recognition.

After the long discussion on how to solve the task with Fourier Transform, the remaining problem is how to calculate the transform. There are several possibilities. You may use an FFT library or implement an FFT algorithm. In case you do not have access to a FFT algorithm/library, you may resort to a slower solution. Instead of performing Fourier transform by a sofisticated algorithm, one may do it directly from the formula

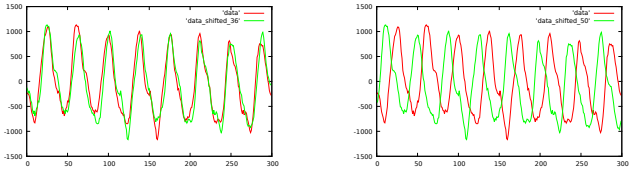
$$result_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i \frac{k}{N} n}$$

(see http://en.wikipedia.org/wiki/Discrete_Fourier_transform for details). The straighforward algorithm is quadratic but wait! We do not need to compute the whole sequence – we know tone frequencies so we just need to calculate results for them. Moreover, in this notation it is possible to calculate the results even for non-integer values of k so we may get a bit more of a precision.

The Fourier transform is way too complicated (and not necessary)

There is a much simpler solution to the problem. The basic idea is this: If we know the frequency of the tone, we know the period of our data. And if we have a period, we can somehow verify whether it is correct. One such simple verification technique: take the sequence, take its copy shifted by the assumed period, and check whether they approximately match.

As an example, consider following figure. On the left, the guessed period is 36 samples corresponding to D#5 (622 Hz). On the right, the guessed period is 50 samples corresponding to A4 (440 Hz). Looking at the pictures, it is clear which tone is the correct one.



However, there is a small catch. If some period T works, the signal will also be periodic with periods $2T$, $3T$, and so on. This is the opposite effect as with harmonics. Of course, we don't care about the octave, we should be safe. So where is the catch? In music it is called a perfect fifth and it happens if we have two tones which are 7 half-tones apart. The ratio of their frequencies is $2^{(7/12)} = 1.4983$ which is roughly 3:2. Thus, adding one octave, we have ratio of roughly 3. An example of such pair of tones is A3 (220 Hz) and E5 (660 Hz). Thus, in this case, our algorithm might misclassify E5 as A3. One solution is to boost preference for higher tones, for example by weighting the resulting correlation error by the period.

Conclusion: This task was about playing with the data and a little bit of tweaking. If you were not sure about the result, one good way to check it was to use some good instrument for checking. Such an instrument are your ears. For example, after classifying all the tones, one can concatenate all recognized tones of the same frequency, convert it to wav file and play it. If you made a mistake, you will hear it instantly!

Problem K: Keys and locks

To: Agent 047.
Mission: Acquire suitcase from room #723.
Briefing:
We managed to get you into room #719, which is on the same floor of the hotel. The hotel still uses the old pin tumbler locks. Housekeeping should have a master key that works for the entire floor. Obtain it in order to get to room #723 undetected. This mission statement will self-destruct in five seconds. In case it does not, please tear it into small pieces and eat it.

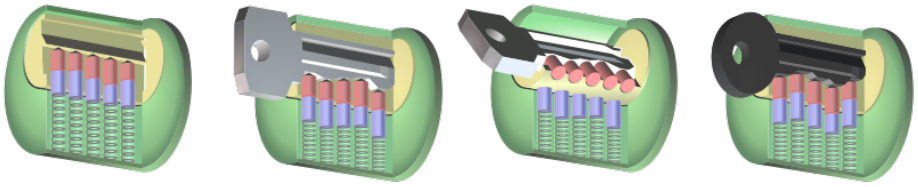
Pin tumbler locks

The photograph on the right shows a door with a pin tumbler lock. The static part of the lock is highlighted in green. The inside part, called the plug, is shown in yellow. This part of the lock rotates when a correct key is inserted and turned.

Below are four drawings of the inside of the lock. The first one shows a lock without any key. Inside the lock there are multiple metal pins (blue+red). If there is no key present, the springs push these pins almost, but not entirely, into the plug.

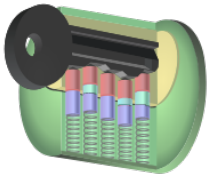
The second and third drawing show what happens when a correct key is inserted into the lock: Each of the pins is actually cut into two separate parts (a blue part and a red part). When a key is inserted into the lock, it pushes the pins out of the plug. The correct key has the property that the boundary between the red and the blue part of each pin exactly matches the boundary of the plug. When this happens, the key can be used to turn the plug and thereby to lock/unlock the door.

The last drawing shows what happens when an incorrect key is inserted into the lock. The pins will be aligned differently and they will block the plug from turning.



Master keys

When you run a hotel, you have the following situation: On one hand, you have a lot of guests, and each guest should only be able to open their own door. On the other hand, you don't want housekeeping to carry a ring with hundreds of keys. It is way more convenient for them to have a *master key* – a single key that can open all the locks.



With pin tumbler locks, creating a master key is really simple. You can even take a set of existing locks+keys and one new key and modify the locks to achieve the desired state – each lock can still be unlocked by its original key, and the new key can unlock all of the locks. How to do this? Each pin in each lock will now be divided in *two places*: one that corresponds to the original key, and one that corresponds to the new master key. (Technically, some of the pins will still be divided in one place. This happens whenever the original key and the master key have the same height at the corresponding location.)

The drawing on the right shows the lock from the previous drawings modified to also admit the second key. Three of the five pins are now divided into three parts (blue, cyan, and red). Now the plug can also be rotated with this new key inside. When that happens, the red and some of the cyan parts of the pins will be inside the plug.

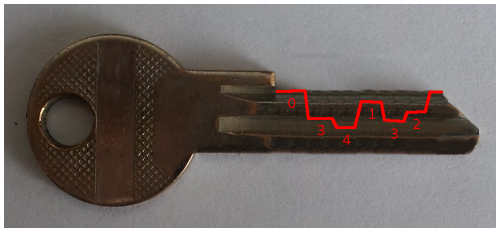
Key profiles and bitting

On the right you can see a *key profile* – an "empty" key. If you want to create a new key for a lock, you need to take the corresponding key profile (one that fits your lock) and then you need to create the right pattern (called *bitting*) on the part of the key that goes into the lock. The locksmiths have machines that do this easily, but you can also do it using an iron file, if you have to. (In this task, you have to.)

Each particular bitting can be described by a string of digits. If you have a lock with p pins, there will be exactly p locations on the key that touch the pins when the key is inserted properly. Each pin can be cut in one of $k \leq 10$ places. (The possible cut locations are sufficiently apart to prevent the same key from working with differently cut pins.) Thus there are exactly k^p different lock-key pairs. Each particular key can be described as a string of p digits from the set $\{0, \ldots, k-1\}$.

Different manufacturers use different encodings of the bitting. In this problem, 0 represents the shortest pin and correspondingly the shallowest cut on the key. (I.e., the larger the number, the more you have to file away when creating the key from an empty profile.) The first number is the pin closest to your hand. (This actually does not matter in this problem.)

In this problem, an empty key profile works as a key with bitting containing all zeroes.



(Note: In practice there is also a standard saying that adjacent pins should have similar cut locations. This is to ensure that the correct key can be inserted and removed smoothly. In this problem we will ignore this and consider all possible keys valid – after all, you will not be using any key too many times.)

The red line in the figure on the right shows how to file the key profile in order to obtain a key for $p = 6$, $k = 5$, and bitting 034132.

Problem specification

You may assume that the bittings of the three actual keys (the master key and the guest keys for rooms #719 and #723) are distinct and that they were generated uniformly at random before you started solving the task. (I.e., our grader will not try to do any nasty tricks.)

Your task is to unlock the room #723, using any key that works. There are two completely independent scenarios with slightly different rules. You will be submitting your actions for the easier scenario as output files for the subproblem K1, and for the harder scenario as K2. Here is the list of allowed actions:

- "`checkin`" – check into room #719 at the hotel. Our answer will have the form: "`room key: XXXXX`", where the string of Xs will be replaced by the bitting of the guest key to room #719. (You are not allowed to modify this key, you will need it to check out without raising any suspicions.)
- "`file A XXXXX`" – use an iron file on key profile number A to produce the specified bitting. Our answer will have the form "`key A new bitting YYYYY`". Note that `YYYYY` may differ from `XXXXX`. We will only apply the changes that are still possible. E.g., if you take a key with bitting 444222 and try to file it to the bitting 333333, you will get the key 444333.
- "`try A R`" – try using key A to unlock room R. The room number must be one of 719 and 723. Our answer will have the form "`success`" or "`failure`". Once you get the answer "`success`" for $R = 723$, you have successfully solved the particular subproblem.
- "`restart`" – only use this if you did something stupid and want to start again from scratch. We will generate new keys for you and reset all your key profiles. Our response will be "`restarted`".

Rules common for both subproblems

- You are allowed to make at most **30 submissions** (not just 10 as for other problems).
- Each submission in which you do not solve a subproblem does count as an incorrect submission. (Using as few submissions as possible is probably not a bad idea.)
- Each submission must be a text file containing between 1 and 20 lines, inclusive.
- Each line must contain exactly one of the commands specified above.
- Commands are processed one by one in the order in which you specified them.
- Lines with incorrect syntax are ignored and an error message is printed for each of them.

Specific rules for the easy subproblem

- The number of different pin heights is $k = 3$.
- The number of pins in the lock is $p = 9$. That means there are $3^9 = 19\,683$ possible keys. (Two of those are the master key and the guest key to #723.)
- At the beginning, you have 160 empty key profiles, numbered 1 through 160. (In other words, you have 160 keys, each with the bitting 000000.)
- You have the additional (possibly useless) information that the bittings for the master key and for the guest key to room #723 differ in all positions. (This does not have to be the case for the key to room #719.)

Specific rules for the hard subproblem

- The number of different pin heights is $k = 9$.
- The number of pins in the lock is $p = 10$. (There are $9^{10} = 3\,486\,784\,401$ different keys.)
- At the beginning, you have 23 empty key profiles, numbered 1 through 23.
- Trying to unlock the room #723 is dangerous and requires a lot of time. The command "`try A 723`" may only be sent *as the only command in the file*. That is, the file must contain just a single line with this command. In all other cases this command will be ignored with an error message.

Communication example (for the easy subproblem)

First submission

```
checkin
file 13 012012012
try 13 723
file 13 111111111
try 9999 -47
```

Our response

```
room key: 202102022
key 13 new bitting 012012012
failure
key 13 new bitting 112112112
try failed -- incorrect syntax
```

Second submission

```
try 13 719
restart
checkin
file 13 012012012
try 13 723
```

Our response

```
failure
restarted
room key: 220012110
key 13 new bitting 012012012
success
```

Notes: The problem statement describes an actual way how master keys for pin tumbler locks are made.
The first four drawings in the problem statement are derived works from pictures in the Wikipedia article "Pin tumbler lock".
Their authors are Wikipedia users Wapcaplet, GWirken and Pbroks13. The drawings are available under the CC BY-SA 3.0 licence.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek, Michal 'Mic' Nánási

Solution

The idea for this problem comes from a lovely research paper by Matt Blaze. You can find the paper at http://www.crypto.com/papers/mk.pdf and we encourage you to read it, possibly instead of the rest of this solution.

The punchline is that using master keys for these old pin tumbler locks is really, really, really insecure.

The easy version

The first thing you need to realize is that there will be more than two keys that open the door to room #723. Why? Because all we need is a key that, for each pin, matches one of the positions where the pin is cut. We do not care that some of these cuts were made for the master key and some for the guest key. Any combination will do.

For example, if the bitting for the master key is 215452 and for the guest key 312044, then any of the keys 212042, 315054, and 215444 will work as well (as will 27 other keys).

With the additional information we have in the easy version (the master key and the guest key differ everywhere), this means that instead of just 2 keys, there are exactly 2^9 keys that will unlock room #723 (out of the 3^9 possible keys).

In 30 submissions we may give 600 commands, and thus we are able to try at most 300 keys in the lock of room #723. (It always takes one command to produce a key, and another to check it.)

Now assume that each key we try is picked uniformly at random. For any particular key, the probability that it fails is $1 - (2^9/3^9) \approx 0.9739877$. In other words, in each attempt we have almost a 3% chance of hitting a key that works. That's quite good – already after 39 attempts the *expected* number of successes exceeds 1.

What is the probability of zero successes in all 300 attempts? It's just $(1 - (2^9/3^9))^{300} \approx 0.000368$. That's worth a try, isn't it?

Also, note that you were only given 160 profiles. In order to make 300 guesses you had to reuse some of them. The simplest solution is to generate 300 random keys, *sort them according to their bitting* and greedily reuse a profile whenever possible. This will get you under 160 used profiles with high probability.

The above approach has one more advantage – it avoids trying the same key twice. This reduces the chance of failure to approximately 0.000346.

Of course, there are better solutions than the one we just presented. In particular, the algorithm that solves the hard version can also be applied to the easy version.

The hard version

We already know one bad property of this master key system – there are actually many other keys that work where they should not. But don't worry about that, now it gets much worse. We will now show that the system has almost no security. In the setting from the problem statement it is possible to actually determine the master key very efficiently.

The trick is very simple – just find its bitting one digit at a time.

In the first submission we will just check in to obtain our room key.

Suppose that your room key has bitting 314512. Look at the first position. There are two possibilities: either the master key starts with a 3 as well, or it starts with some other digit x. Now, there are just $k-1$ keys with the bitting $x14512$ where $x \neq 3$. We may as well try them all in the lock of *our* hotel room. If one of them fits, we know the first digit of the master key. If none of them fits, the first digit of the master key has to be the same as the first digit of our room key.

Additionally, we can do everything above using just a single key profile – just file it down more and more until it fits.

And of course, we can do the same with all other positions. In parallel, using 10 key profiles – one for each position. In each submission we can modify and try using each of the keys.

This means that in just $p(k-1)$ queries (in the worst case!) we know the master key and we can just waltz into any room we like. Crazy, huh? Instead of over three billion possible keys, you just try at most 80 keys in your own lock, and suddenly you have the master key.

(As the key was generated at random, you should need less than 8 submissions – once you know some digit of the master key, you can look for another digit twice as fast by using two profiles for the same position. Still, it may be a better strategy to make submissions 2 through 9 in parallel, trying all 80 keys. You will get more penalty minutes for this task, but you'll have more time to solve other tasks and thus score more points.)

Final notes

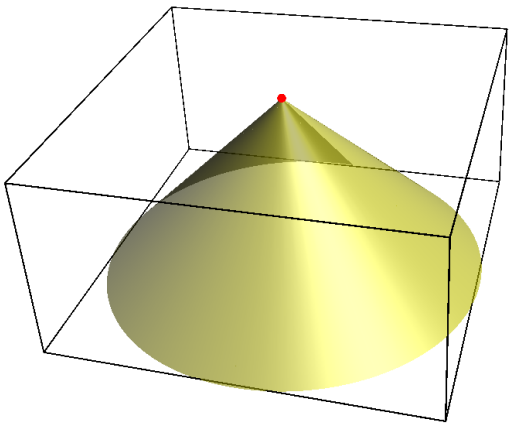
The constraint that the room key and master key differ on all locations in the easy subproblem served two purposes. First, it allowed us to use a larger key space and still make it possible to randomly guess the correct key. Second, it eliminated the luck factor. If we did not include this constraint, some teams would have a bigger chance of guessing a good key than others.

In practice this constraint is not that artificial – for locks that have approximately $k = 10$ it is very likely that the master key differs from a particular room key in almost all positions.

Problem L: Light in a room

It was a tough year for Per. After finishing his post-doc and going to a lot of interviews, he finally got a job at a university on the other side of the country and had just 2 weeks to find a new apartment. In the end he managed to find one – a really old and dusty apartment. He spent the whole weekend cleaning, throwing away old stuff and repainting the walls. On Sunday evening after 12 hours of work, he couldn't take it anymore. He just rested on the floor, covered in paint and dust.

The room he just painted was empty, only a lamp was mounted to the ceiling. As he switched it on, he noticed that the light rays coming from the lamp formed a cone and covered some parts of the floor and walls. All the work exhausted his body, so he couldn't move, but his mind was still working and wanted to solve some hard problems. He started wondering about the area covered by the light from the lamp.



Obr. 1: An example of light emitted by a lamp.

Problem specification

The room has a horizontal floor, a horizontal ceiling, and vertical walls. The floor is a **convex polygon**. (In the easy data set the polygon is an axes-parallel **rectangle**.) There is a lamp mounted somewhere on the ceiling of the room. The lamp emits a cone of light downwards. The axis of the cone is vertical.

You are given the height of the room, the description of the floor, the location of the lamp and the angle at the apex of the cone of light emitted by the lamp.

Your goal is to calculate the total area of all lit surfaces in the room. (One of these surfaces will always be on the floor. There may be additional lit surfaces on some of the walls.)

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a series of lines. The first line contains space separated floating point numbers l_x, l_y, h, and α. The first two numbers are the horizontal coordinates of the lamp. The third number is the height of the room (and at the same time the vertical coordinate of the lamp). The last number is the angle at which the lamp emits light. (For any ray of light from the lamp, the angle between the ray and the axis of the cone is at most $\alpha/2$ degrees.)

The second line of a test case contains the number n of vertices of the floor. Each of the next n lines contains 2 floating point numbers x_i, y_i – the coordinates of the i-th vertex of the floor. The coordinates are given in counterclockwise order. The z-coordinate of the floor is 0.

In the easy data set in each test case we have $n = 4$ and the polygon is an axes-parallel rectangle.

In the hard data set in each test case we have $n \leq 100$ and the polygon is convex.

Output specification

For each test case output one line with one floating point number – the total area of all surfaces that are directly reached by the light of the lamp. Output at least six decimal places. Solutions with a relative or absolute error at most 10^{-6} will be accepted.

Example

input	output
1 5 5 5 91 4 0 0 10 0 10 10 0 10	81.320740643

Task authors

Problemsetter: Lukáš 'lukasP' Poláček
Task preparation: Lukáš 'lukasP' Poláček, Michal 'mišof' Forišek

Solution

Let $L = (l_x, l_y, h)$ be the position of the lamp and let $K = (l_x, l_y, 0)$ be its projection onto the floor. We can divide the room surface into n parts by line segments from K to all vertices of the floor. That is, each part contains a triangle of the floor (determined by the point K and one side of the room) and one wall.

Light on the floor

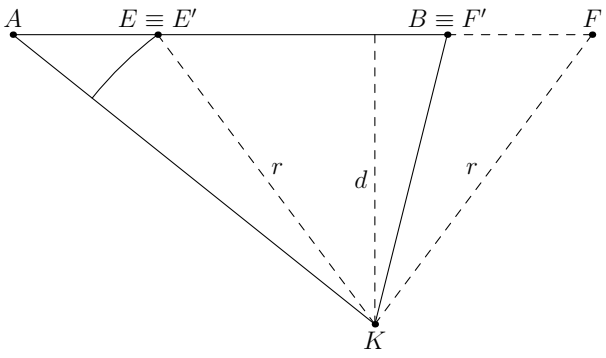
If there were no walls, the part of the floor covered by light would have the shape of a circle. Let us denote this disc as D. The center of D is obviously K, and we can compute its radius as $r = h \tan \alpha/2$. Once we add the walls, the lit part of the floor becomes the intersection of D and the floor.

To compute the area of this intersection, we can consider each triangle of the floor separately. Note that computing the intersection of a floor triangle and our disc D is quite easy, as the center of D is one of the vertices of the triangle. This significantly reduces the number of cases to consider.

Let ABK be the triangle on the floor. Let us rotate the triangle around A so that A and B share the same y-coordinate. This can be done easily with complex numbers. For a moment, we consider A, B, K as numbers in the complex plane. Let $v = (B - A)/|B - A|$, i.e., a number with the same direction as $B - A$ but of length 1. The new coordinates of B and K are $(B - A)/v + A$ and $(K - A)/v + A$. For more information we refer to an excellent tutorial by Bruce Merry on TopCoder: Computational geometry with complex numbers.

Furthermore, let us shift the triangle so that K moves to coordinates $(0, 0)$. Thanks to the rotation and shift, the distance d from K to the line AB is $|A_y| = |B_y|$, i.e. the absolute value of the common y-coordinate of A and B.

What if d is at least as big as the radius r? The disc D does not intersect the line AB, at most it may touch it. In that case the lit area on the floor is a circular sector with angle AKB. In particular, if the angle AKB is φ radians, the area on the floor is $(\varphi/2)r^2$. Also, in this case there is surely no light on the corresponding wall.



Obr. 2: An example of a situation on the floor.

From now on we assume that $d < r$. Now the intersection of the line AB and the disc D is a line segment EF, where $E_x = -D_x, F_x = \sqrt{r^2 - d^2}$ and $E_y = F_y = A_y$. This follows from the fact that E, F lie on AB and the distance from K to the line is d. We use Pythagorean theorem to derive E_x and F_x.

If the segments EF and AB are disjoint, then the disc D does not intersect the line segment AB and the lit area is the same circular sector as above.

In the remaining case let $E'F'$ be the intersection of segments EF and AB. (The segment $E'F'$ is the lit part of the bottom of the current wall.) The intersection of disc D and triangle ABK can now be divided into three parts: the circular sector with the angle AKE', the triangle $KE'F'$, and the circular sector with the angle $F'KB$. (Each of the two circular sectors may be empty – this happens if $E' = A$ or $F' = B$, i.e., when a corner of the room is lit.) We already know how to compute the area for the circular sectors, and we can easily compute the area of the triangle $KE'F'$ as $|E'F'| \cdot d/2$.

Light on the walls

Calculating the area lit on the walls is the harder part of this problem. Similar to calculating the area on the floor, we rotate and shift everything so that $A_y = B_y$ and $K = (0, 0)$. The wall is lit only when EF and AB have a non-empty intersection, so from now on we assume that this is the case. This implies that there exist points E' and F' as defined above.

The are lit on the wall is an intersection of a cone and a plane, thus it is a conic section. Since the walls are vertical, the area lit on the wall is bounded by a hyperbola. For more information and pictures see articles on Conic section and Hyperbola on Wikipedia.

For clarity, in the next part we will use the s-axis and the t-axis to refer to the points on the wall. The s-axis is parallel to the bottom of the wall (so after the rotation we did above, the s-axis is just the x-axis). The t-axis is parallel to the z-axis. The point $(0, 0)$ is the orthogonal projection of the lamp onto the plane of the wall and it is also the center of the hyperbola. Note that t points upwards, so the projection of K onto the wall has coordinates $(0, -h)$.

In our case the hyperbola is of the form $(s/a)^2 - (t/b)^2 = -1$ for some a and b that we shall determine later. Since we are only interested in the part of the curve that is on the wall, we only consider the lower part of this curve, namely the one with negative t. We can write

$$t = f(s) = -b\sqrt{(s/a)^2 + 1}.$$

By the definition of E' and F', these are the two points where the hyperbola on the wall "starts" and "ends". Let c, d be the s-coordinates of E' and F' respectively. In other words, no point under the hyperbola has s below c and above d.

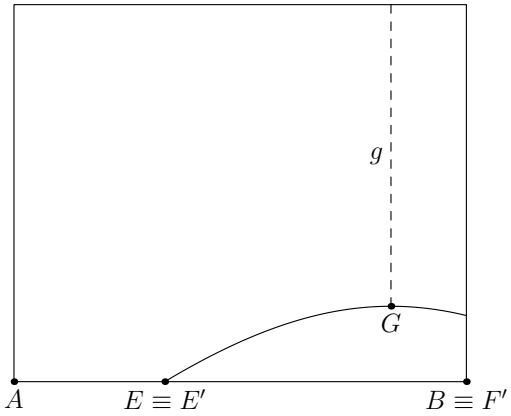
The area lit by the lamp is then given by the integral $\int_c^d (h + f(s))\,\mathrm{d}s$. We have

$$\int_c^d (h + f(s))\,\mathrm{d}s = (d - c)h - \frac{b}{2}\left[s\sqrt{\frac{s^2}{a^2} + 1} + a\sinh^{-1}\left(\frac{s}{a}\right)\right]_c^d.$$

The only remaining part is to calculate a and b. If we find 2 points which are on the hyperbola with known coordinates, then we can determine these two parameters. Let $G = (0, -g)$ be the point of the hyperbola closest to the lamp. A simple application of the rule of similar triangles gives us $g = hd/r$. Since the hyperbola passes through this point, we have $g/b = 1$ and thus $b = \pm g$.

For a, we can use points E and F (as defined above), since we know they lie on the hyperbole. We have $(E_x/a)^2 - (-h/b)^2 = -1$, which implies

$$a = \frac{E_x}{\sqrt{(h/b)^2 - 1}}.$$



Obr. 3: An example of a situation on the wall.

Using $b = hd/r$, this can be further simplified to

$$a = \frac{E_x}{\sqrt{(r/d)^2 - 1}}.$$

Problem M: Matrix nightmare

As Obi-Wan would put it: "This isn't the problem statement you are looking for. Move along."

The *double factorial* numbers are the numbers d_i defined by the following recursive formula: $d_0 = 1$, and $\forall i > 0 : d_i = d_{i-1} \cdot i!$. For example, $d_3 = 1! \cdot 2! \cdot 3! = 12$.

Sequences of length n with all elements belonging into the set $\{0, \ldots n-1\}$ are called *limited sequences of order* n. For example, $(0, 2, 0, 1)$ is a limited sequence of order 4. The set of all limited sequences of order n will be denoted $\mathcal{S}_n$.

The *spread factor* of a sequence $A = (a_0, \ldots, a_{n-1})$ is the value $\sigma(A) = \prod_{i=0}^{n-1} \prod_{j=i+1}^{n-1} (a_i - a_j)$.

There is a direct isomorphism between pairs of sequences and sequences of pairs. Formally: Let $A, B \in \mathcal{S}_n$. We can denote their elements as follows: $A = (a_0, \ldots, a_{n-1})$, $B = (b_0, \ldots, b_{n-1})$. The corresponding sequence of pairs $\big((a_0, b_0), \ldots, (a_{n-1}, b_{n-1})\big)$ will be denoted $P_{A,B}$.

Pairs of integers can be ordered lexicographically in the usual fashion: $(a, b) \leq (c, d)$ if either $a < c$, or $(a = c \wedge b \leq d)$. A sequence $P = (p_0, \ldots, p_{n-1})$ of pairs of integers is ordered lexicographically if for all i, $p_i \leq p_{i+1}$. Let $\rho(P) =$ [if P is ordered lexicographically then 1 else 0].

Let M be a $n \times n$ matrix, with rows and columns indexed from 0 to $n-1$. Elements of M will be denoted $m_{r,c}$. A matrix is called a *$\mathbb{Z}$-var matrix* if each element in the matrix is either an integer or a variable. The *n-step traversal weight* of M is the following value:

$$\varphi(M) = \frac{1}{d_{n-1}^2} \cdot \sum_{\substack{A=(a_0,\ldots,a_{n-1}) \\ A \in \mathcal{S}_n}} \sum_{\substack{B=(b_0,\ldots,b_{n-1}) \\ B \in \mathcal{S}_n}} \left(\rho(P_{A,B}) \cdot |\sigma(A)| \cdot \sigma(B) \cdot \prod_{i=0}^{n-1} m_{a_i,b_i} \right)$$

Problem specification

Given is a multivariate polynomial p with integer coefficients. Produce any reasonably small $\mathbb{Z}$-var matrix M such that $\varphi(M) = p$.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line describing the polynomial. The variables are the letters `a` through `z`, the syntax will be clear from the input file. Each polynomial in the input file contains less than 50 operations (i.e., additions, subtractions and multiplications).

Output specification

For each test case output one matrix in the following format: First its size n, then all its elements in row major order. The elements may be separated by any positive amounts of whitespace. The size of the matrix must not exceed 70. All integers must be between -10^9 and 10^9, inclusive.

If for a given polynomial no such matrix exists, output a single zero instead.

Example

input	output
1 xy	2 1 y x 0

Task authors

Problemsetter: Michal 'mišof' Forišek
Task preparation: Michal 'mišof' Forišek, Lukáš 'lukasP' Poláček

Solution

The first step is to simplify the scary problem statement. It basically reduces to "find a matrix M such that the determinant of the matrix is the given polynomial". Below we first explain why this is the case, and then solve this simplified problem.

Dissecting the problem statement

First let's take a look at the limited sequences and their spread factors. Recall that the *spread factor* of a sequence $A = (a_0, \ldots, a_{n-1})$ is the following value: $\sigma(A) = \prod_{i=0}^{n-1} \prod_{j=i+1}^{n-1} (a_i - a_j)$. By definition, $\sigma(A)$ is the product of a bunch of numbers. And as soon as one of those numbers is 0, the entire product becomes zero. In other words, $\sigma(A)$ is non-zero if and only if all elements of A are distinct.

Okay, but if $\sigma(A) \neq 0$ and A is a limited sequence, we only have n distinct values that may appear in A. This means that each of them must appear exactly once – and therefore A is a permutation of the set $\{0, \ldots, n-1\}$.

And what is $\sigma(A)$ if A is such a permutation? For all possible permutations we will get exactly the same value. This is because for each pair of *values* we take their difference exactly once. The only difference will be the sign – for some A, $\sigma(A)$ will be positive, for others it will be negative.

For example, for any permutation of $\{0, 1, 2, 3\}$ we would see six factors in the product: $(3-0)$, $(3-1)$, $(3-2)$, $(2-0)$, $(2-1)$, and $(1-0)$ – but each of them can be in the other order, thereby flipping the sign of the result.

What's the absolute value of $\sigma(A)$ in this case? Among the factors we have once $n-1$, twice $n-2$, and so on. Hence $|\sigma(A)| = d_{n-1}$. (The value d_{n-1} is the double factorial from the problem statement.

And what is the sign of $\sigma(A)$? The sign flips each time $a_i < a_j$ for some $i < j$. If we denote the number of inversions of A as $\text{inv}(A)$, then we can write $\sigma(A) = (-1)^{n(n-1)/2 - \text{inv}(A)} d_{n-1}$.

Knowing this, we can now take a look at the final formula:

$$\varphi(M) = \frac{1}{d_{n-1}^2} \cdot \sum_{\substack{A=(a_0,\ldots,a_{n-1}) \\ A \in \mathcal{S}_n}} \sum_{\substack{B=(b_0,\ldots,b_{n-1}) \\ B \in \mathcal{S}_n}} \left(\rho(P_{A,B}) \cdot |\sigma(A)| \cdot \sigma(B) \cdot \prod_{i=0}^{n-1} m_{a_i,b_i} \right)$$

We immediately see that some of the terms in the sum will be zero. Actually, quite many of them will be zero. For some of them the reason will be $\sigma(A) = 0$ or $\sigma(B) = 0$. Those two are non-zero if and only if A and B are both permutations.

For others the reason will be $\rho(P_{A,B}) = 0$. To avoid this, A (which is a permutation) must be sorted in ascending order.

In other words, $\rho(P_{A,B}) \cdot |\sigma(A)| \cdot \sigma(B) \neq 0$ implies that $A = (0, \ldots, n-1)$ and that B is an arbitrary permutation. We also know that $|\sigma(A)| = d_{n-1}$ and that $\sigma(B) = (-1)^{n(n-1)/2 - \text{inv}(B)} d_{n-1}$.

Let $\mathcal{P}_n$ denote the set of all permutations in $\mathcal{S}_n$.

This allows us to rewrite the main sum in the following form:

$$\varphi(M) = (-1)^{n(n-1)/2} \cdot \sum_{\substack{B=(b_0,\ldots,b_{n-1}) \\ B \in \mathcal{P}_n}} (-1)^{\mathrm{inv}(B)} \cdot \prod_{i=0}^{n-1} m_{i,b_i}$$
$$= (-1)^{n(n-1)/2} \cdot \det(M)$$

Hence we are solving the task "given is a polynomial, find a matrix such that the polynomial is its determinant". (To deal with the sign, we then just append a few rows and columns of zeroes, with 1s on the diagonal, to get a matrix with the same determinant and n divisible by 4.)

Constructing the matrix

Getting this far should be sufficient to solve the easy data set by hand. For example, here are matrices with determinants equal to abc, $2a^3$, $a+b$, and $a-b$, respectively:

```
3              4              2          2
a 0 0          2 0 0 0        a b        a b
0 b 0          0 a 0 0        -1 1       1 1
0 0 c          0 0 a 0
               0 0 0 a
```

(Pad these to 4×4 to get answers for the original problem.)

In general, it is easy to create matrices where the determinant is a product of terms – just throw all terms onto the diagonal and pad the rest with zeroes.

What we still need to find out is how to do addition. For instance, assume I have the polynomial $ae + bcd$. I can easily construct matrices that correspond to ae and to bcd, respectively. Is there a mechanical transformation that will take those two matrices and produce a new one for $ae + bcd$?

This is indeed possible, and with some experimentation one can come up with the necessary transformations. However, in this sample solution we would like to give you a little bit of additional insight, so we will show a nice systematic way of coming up with the construction.

The solution presented below was discovered and published by Leslie Valiant in the paper Completeness Classes in Algebra (STOC 1979). By the way, this paper and his work on the #P complexity class were some of the major reasons why he won a Turing Award in 2010.

Directed graphs and cycle covers

Instead of working directly with matrices it will be more convenient to imagine expressions as (some special) directed graphs. All operations (additions and multiplications) will translate into operations on graphs. Only in the end will we convert the graph that represents the entire polynomial into a corresponding matrix.

Why graphs? Determinants and permutations are closely related to *cycle covers*. If we have a directed graph on n vertices, a cycle cover is a set of n edges such that each vertex has precisely one incoming and one outgoing edge in the set. In other words, these n edges form one or more cycles that together cover all vertices in the graph – hence the name.

In a complete directed graph (with loops), each permutation of the vertices determines a cycle cover and vice versa. (The cycles of the permutation are precisely the cycles in the cover.)

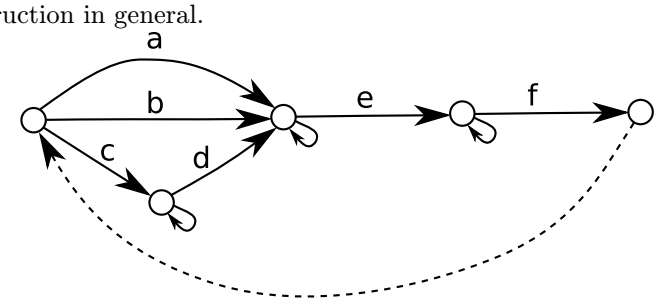
If you now consider a $n \times n$ matrix, you can view the elements as edge weights. For example, if $m_{0,3} = 47$, we interpret this as "the edge from 0 to 3 has weight 47". It may be convenient to interpret $m_{i,j} = 0$ as "the edge from i to j is not present at all".

Now consider the definition of the determinant again:

$$\det(M) = \sum_{\substack{B=(b_0,\ldots,b_{n-1}) \\ B \in \mathcal{P}_n}} (-1)^{\mathrm{inv}(B)} \cdot \prod_{i=0}^{n-1} m_{i,b_i}$$

The sum goes over all permutations B, in other words, over all cycle covers of the $n \times n$ graph determined by M. Now, for a particular cycle cover B the product $\prod_{i=0}^{n-1} m_{i,b_i}$ is just the product of weights of all n edges that form the cycle cover. We will call this value the weight of that particular cycle cover.

So, basically what we want to do: given the polynomial, we want to construct a graph such that the sum of its cycle cover weights is the polynomial. One technical issue we will have to deal with are the plus/minus signs – when computing the determinant, we add the cycle covers determined by even permutations, and subtract cycle covers determined by odd permutations. As we shall show below, the easiest way to handle this technicality is to construct the graph in such a way that all its cycle covers correspond to even permutations. This will always be possible. We will first show you an example and then explain the construction in general.



Consider the graph above. Let's call the leftmost node the source and the rightmost one the sink. First, note that there are exactly three source-sink paths: aef, bef, and $cdef$.

Now, how do the cycle covers of this graph look like? Clearly, the cycle that contains the source must consist of a source-sink path and the dashed edge. And once we choose that cycle, we will not be able to form any other non-trivial cycles. So if there are still some vertices not on the same cycle as the source and the sink, each of them forms a separate single-vertex cycle.

If we assume that the dashed edge and the loops all have weight 1, the above graph has exactly 3 cycle covers and their weights are aef, bef, and $cdef$. In other words, the graph is pretty close to representing the polynomial $(a + b + cd)ef$.

We just have to fix the signs. If we computed the actual determinant of the weight matrix of this graph, we would not get $+aef + bef + cdef$. We would get $-aef - bef + cdef$. This is because the cycles corresponding to aef and bef have an even length, so their corresponding permutations have an odd number of inversions.

There is a simple way to fix this – we will just add some vertices to make sure that each source-sink cycle contains an odd number of nodes.

Constructing the graphs

We will first transform any polynomial into a directed acyclic graph with the following properties:

1. One of the vertices is labeled as a source.
2. One of the vertices is labeled as a sink.
3. Each edge is labeled by a variable or a constant.
4. Let the weight of a source-to-sink path be the product of labels on it. Then the sum of these weights

over all source-to-sink paths is equal to our polynomial.
5. The lengths of all paths from source to sink have the same parity.

(After this construction is done, we will tweak the graph to make it cyclic and to have the required cycle covers. The last property is needed to make sure that all cycles containing the source and the sink will have an odd length.)

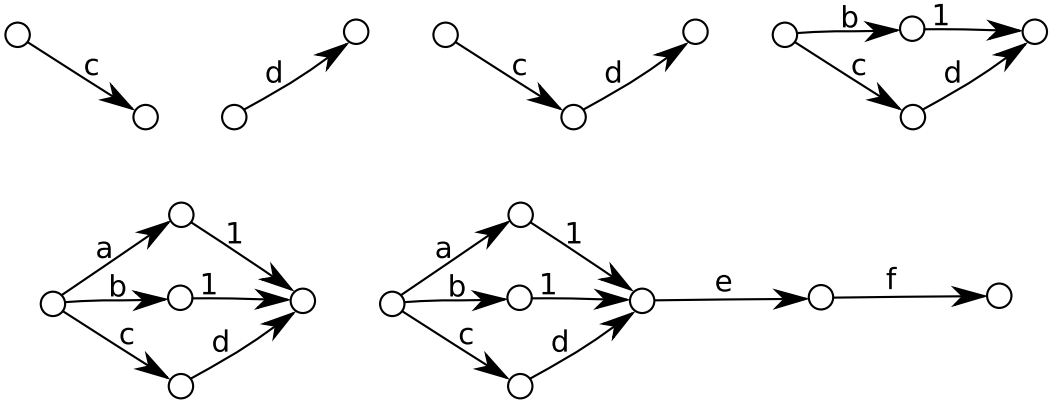
The construction will be pretty simple – it follows the intuition we already have from the above example.

- The graph for a constant or a variable consists of the source, the sink, and one edge with the corresponding label.
- If the polynomial p can be written as $q \cdot r$, the graph G_p is constructed by constructing the graphs G_q and G_r, and then merging them by identifying the sink of G_q with the source of G_r.
- If the polynomial p can be written as $q + r$, we also start by constructing the graphs G_q and G_r. Now we have two cases.

  If all source-sink paths in G_q have the same parity as in G_r, we identify the source nodes of both graphs, we add a new sink node, and edges of weight 1 from the sink nodes of G_q and G_r to the new sink node.

  If all source-sink paths in G_q have the opposite parity as in G_r, we identify the source nodes of both graphs and we add an edge from the sink of G_q to the sink of G_r. The sink of G_r will be the sink of the new graph.

The pictures below show the graphs that correspond to the polynomials c, d, cd, $b + cd$, $a + b + cd$, and $(a + b + cd)ef$. (Note that the result is not the only graph that represents $(a + b + cd)ef$. If we did the construction in a different order, we could have obtained a slightly different graph with the same properties.)



To finish the construction: All nodes other than the source and the sink get loops of weight 1. If the paths from source to sink have an odd number of vertices, we add an edge of weight 1 from the sink to the source. Otherwise we identify the sink and the source.

