IPSC 2014

problems and sample solutions

Problem S: Say it loudly!

Alice was walking down the street. When she saw Bob on the other side, she shouted: “HELLO, BOB!”

“Hi, Alice!” responded Bob from afar and they both hurried to meet in the middle of the street.

“Bob! Why didn’t you shout back when I said hello?” asked Alice, apparently a bit annoyed.

Bob, surprised, defended himself: “What? But I did. I said ’Hi, Alice!’”.

Alice smiled. “Oh, I see. You were too quiet. To shout, you must speak in uppercase!”

“Ah, OK. No, wait. I said ’Hi’, which has an uppercase letter, see?”

“That makes no difference. The whole word must be in uppercase, LIKE THIS. Words that are entirely in uppercase are twice as loud as normal words.”

“Your shouting is only twice as loud as normal?” laughed Bob. “That’s not very much.”

“Just you wait!” responded Alice and produced a terrible high-pitched squeal: “*Eeeeeeeee!*”

“How did you do that?”

“You just have to add asterisks. All words between them are three times louder.”

“Cool. Let’s see who can shout louder!” proposed Bob and screamed: “*AAA aaa* a a aargh!”

Alice countered: “*aaaaa*!”

“I won!” exclaimed Bob. “I used three normal words, one word that was thrice as loud, and one that was six times as loud. You could say I scored 1+1+1+3+6=12 loudness points.”

“No, you’ve got it all wrong. You have to count the average, not the sum. So you only scored 2.4 points. And I scored 3, so I won!” concluded Alice.

You are given several sentences. Your task is to find the loudest one.

Input and output 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 an integer n ($1 \leq n \leq 100$), the number of sentences. Each of the following n lines contains one sentence. Each line is at most 500 characters long.

A *word* is a maximal non-empty sequence of consecutive upper- and lowercase letters of the English alphabet. Words in a sentence are separated by spaces ( ), punctuation (`,.:;’"!?-`) and asterisks (`*`). No other characters appear in the input. Each sentence contains at least one word. There is an even number of asterisks in each sentence. The input file for the **easy subproblem S1** contains no asterisks.

For each test case, output one line with a single integer – the index of the loudest sentence (i.e., the one whose average word loudness is the highest), counting from 1. If there are multiple sentences tied for being the loudest, output the smallest of their indices.

Example

input

```
2

2
HELLO, BOB!
Hi, Alice!

4
*AAA aaa* a a aargh!
*aaaaa*
*note*that asterisks do not*nest***
* * THIS IS NOT BETWEEN ASTERISKS * *
```

output

```
1
2
```

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Martin 'Imp' Šrámek
Quality assurance:	Peter 'PPershing' Perešíni

Solution

This was the easiest problem of the practice session. In each test case, you were given a number of sentences and had to compute their average loudness. You would keep the index of the loudest encountered sentence and output it when you have processed the whole testcase.

Each line contained one *sentence*, which was defined as a sequence of *words* separated by spaces, punctuation and asterisks. Note that punctuation contained symbols such as dot (`.`), so our definition of sentence is a bit different from what we know from natural language (since in natural language, dot marks the end of a sentence). Let this not confuse you. We enumerated the punctuation symbols that may be used in the input, but you may notice that this was unnecessary. Since a *word* was defined as a maximal sentence of alphabetical symbols, we may simply treat any non-alphabetical symbol as a word-separator.

We will now show how to process a sentence and compute its average loudness. By the usual meaning of the word 'average', this is the sum of individual words' loudness divided by the number of words. Therefore, we will start by defining two variables `line_sum_loudness` and `word_count` and initialize them to zero. Then, we will scan the line from the beginning to end, one character at a time. For every character, we will check if one or more of the following situations occur.

1. **This character is alphabetical, but the previous one was not.** We are at the beginning of a word. We will define a boolean variable `encountered_lowercase` to remember if we have encountered a lowercase letter in this word yet or not and initialize it to false.
2. **This character is alphabetical.** We are inside a word. We have to update the value of `encountered_lowercase` acccordingly, depending on the case of this character.
3. **This character is alphabetical, but the following one is not.** We are at the end of a word. We will increment `word_count`. If `encountered_lowercase` is false, it means the whole word was in uppercase and its loudness is doubled. For the hard subproblem S2, we must also know whether we are inside a pair of asterisks (we will soon show how to do this). Based on this, we can compute the loudness of the word and add it to `line_sum_loudness`.

Note that the three situations must be checked in order and that more than one of them can occur at a time. Whenever 1 or 3 occurs, 2 must as well. For single-letter words, all three occur at the same time. Also note that in practice, the conditions "previous/following character is not alphabetical" include special cases for the first and last characters of a sentence.

As was mentioned in the computation in situation 3, we may also want to know if we are inside a pair of asterisks. All that is needed is to define a variable `encountered_asterisks`, initialized to false, and then check for one more situation

4. **This character is an asterisk.** Toggle the value of `encountered_asterisks`.

After processing the whole line, we can compute its average loudness as `line_sum_loudness / word_count`. Since this is clearly a rational number (a fraction of two integers), it is advisable not to lose precision by converting it to a floating point datatype, but treat it as a fraction instead. The reason is as follows. As we have to output the index of the loudest line in the test case, we will have to compare the results for individual lines with each other. The easiest way to do this is to remember the loudness of the loudest line encountered so far and then compare it with the result from the currently processed line. Now, doing this comparison, we advise **not** to compare two floating point numbers

```
current_line_avg_loudness = word_total_loudness / word_count
loudest_line_avg_loudness < current_line_avg_loudness
```

Instead, we will compare two fractions

```
loudest_line_sum_loudness / loudest_line_word_count < word_total_loudness / word_count
loudest_line_sum_loudness * word_count < word_total_loudness * loudest_line_word_count
```

where all the four occuring variables are integers. The last line shows how to do the comparison without division, i.e. without the need to leave the domain of integers. You can see that our C++ solver indeed uses this approach. We tried replacing the fractions with the *long double* datatype and luckily we still obtained the correct answer. However, using the *double* or *float* datatypes caused our solver to give a wrong answer for the hard subproblem. This is a problem that some of you may have encountered - and it may have cost you points even if your idea of the solution was right! The moral of this story is - beware of floating point comparisons!

Problem T: Two covers

In a typical genome assembly problem, we are given set of small strings called *pieces* and our task is to find their superstring with some reasonable properties. In this problem, you are given one such superstring and a collection of pieces aligned to that superstring. Your task is to evaluate one particular property.

Problem specification

You are given the length of the superstring ℓ, a list of n pieces, and a number k. The positions in the superstring are numbered from 1 to ℓ. For each piece i you are given the positions (b_i, e_i) of its beginning and end.

The letter at position x in the superstring is *well-covered* if:

- There is a piece (b_i, e_i) such that $b_i \leq x - k$ and $x \leq e_i$.
- There is a **different** piece (b_j, e_j) such that $b_j \leq x$ and $x + k \leq e_j$.

Your task is to count the number of letters which are **not** well-covered.

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 a line containing three integers: ℓ, n, and k. Each of the following n lines contains two integers b_i and e_i ($1 \leq b_i \leq e_i \leq \ell$): the indices of the endpoints of a piece. The pairs (b_i, e_i) are all distinct.

In the **easy subproblem T1** you may assume that $t \leq 20$, $\ell \leq 10\,000$, $n \leq 1000$, and $1 \leq k \leq \ell$.

In the **hard subproblem T2** you may assume that $t \leq 10$, $\ell \leq 10^{17}$, $n \leq 1\,000\,000$, and $1 \leq k \leq \ell$.

Output specification

For each test case, output a single line with a single integer – the number of letters that are not well-covered.

Example

input

```
1

8 3 2
1 4
3 5
4 8
```

output

```
5
```

The well-covered letters are at positions 3, 4, and 5. Note that the letter at position 6 is not well-covered.

Task authors

Problemsetter:	Vlado 'usamec' Boža
Task preparation:	Vlado 'usamec' Boža
Quality assurance:	Michal 'Žaba' Anderle

Solution

The easy subproblem could be solved just by checking whether each position is well-covered. There are at most 10 000 positions and 1000 pieces, so the straighforward $O(n\ell)$ algorithm should work without any problems.

The hard subproblem can be solved by a typical sweeping algorithm. To simplify coding in the hard problem we can employ the following trick: We will read the input and use it to build two new sets of intervals. For each interval $[b_i, e_i]$ from the input, we will add the interval $[b_i + k, e_i]$ into the first set and $[b_i, e_i - k]$ into the second set (but only if these new intervals are non-empty).

Now we can rephrase our task as follows: for every position, check whether there is an interval in the first set and an interval with a different ID in the second set that both contain it.

This can be solved by considering endpoints of intervals as events, ordering the events and processing them sequentialy while maintaining two separate sets of active intervals. This can be done in time $O(n \lg n)$.

Problem U: Urban planning

The town of Pezinok wants to expand by building a new neighborhood. They hired some famous architects to design it, and your friend Jano is one of them. He is in charge of the road network. It is common to make one-way roads these days, so Jano went all out and decided to make *all* the roads one-way. (Of course, a pair of junctions can be connected by two roads – one in each direction.)

Once Jano made a map showing the planned roads, he noticed that some parts of the neighborhood might not be reachable from other parts. To estimate the impact of this issue, he decided to use a systematic approach: he took a piece of paper and wrote everything down. Namely, for each junction j he listed all other junctions that are (directly or indirectly) reachable from j. We call this information the *reachability list* of a road network.

But then Jano's hard drive crashed and he lost all the plans he had made. The only thing he has left is the piece of paper with the reachability list.

Help Jano reconstruct his original road network. Of course, many different road networks can produce the same reachability list. Therefore, Jano asked you to find the smallest possible road network that has the given reachability list. That should help him reconstruct his original plans.

Problem specification

Find a road network with the smallest possible number of roads that has the given reachability list.

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 a line containing an integer n, denoting the number of junctions. The junctions are numbered 1 through n. Next, n lines follow, each containing a string of length n. The i-th of these lines specifies which junctions are reachable from junction i. Namely, the j-th character in the line is 1 if junction j is reachable from i and 0 otherwise. (Note that for each i, junction i is reachable from itself.)

The reachability list is consistent – it describes at least one real road network.

In the **easy subproblem U1** you may further assume that the reachability list is special: for every pair of junctions $a \neq b$, either a is reachable from b or b is reachable from a, but never both.

Output specification

For each test case, output the smallest road network that corresponds to the given reachability list. The first line of the description should contain the number of roads m (which has to be as small as possible). Each of the next m lines should contain two integers a_i and b_i ($1 \leq a_i, b_i \leq n$) such that there is a one-way road going from junction a_i to junction b_i.

You can print the roads in any order. If there are multiple optimal solutions, output any of them.

Example

input

```
2

3
111
011
001

4
1111
1111
0011
0011
```

output

```
2
1 2
2 3

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

The first test case satisfies the additional constraint for the easy subproblem.

In the second test case, we know that all junctions should be interconnected, except that junctions 1 and 2 should not be reachable from junctions 3 and 4. The smallest road network with this property has five one-way roads. One such road network is shown in the example output above. This test case also has other optimal solutions.

Task authors

Problemsetter:	Lukáš 'lukasP' Poláček
Task preparation:	Lukáš 'lukasP' Poláček
Quality assurance:	Ján 'JaNo' Hozza

Solution

Let us first translate the task into graph terminology. Junctions are vertices and one-way roads between them are oriented edges. The reachability list then corresponds to the transitive closure of the graph. Finally, the task is to find the smallest graph with the same transitive closure. We denote the final graph by G.

The first part of our solution is to identify strongly connected components (SCC). Vertices u and v are in the same strongly connected component if and only if u is reachable from v and v is reachable from u. We can make vertices in one SCC reachable from each other by placing them on one big cycle in the final graph G. If there are p vertices in the component, we need p edges to create the cycle. Note that $p-1$ edges are insufficient, since then at least one vertex will have no incoming edge.

After we identified the strongly connected components and placed vertices in each one on a cycle, we take care of the rest of the graph – the edges between SCCs. If we condense the components into vertices, they form a directed acyclic graph (DAG), which we denote by F. Suppose now that for any DAG we are able to find the smallest graph H that has the same transitive closure. For each edge (s, t) in H it is sufficient to put one edge in G from any vertex in component s to any vertex in t and this gives us the smallest graph G with the same transitive closure as the one given in the input.

It remains to figure out how to find H, the smallest DAG that has the same transitive closure as F (the graph given by the transitive closure of the condensed SCCs). We can use an algorithm very similar to Floyd-Warshall algorithm. Suppose u is reachable from v but we can also reach some k from v and then u from k. Since F is a DAG, having an edge from v to u in H would be redundant, since H must also have k reachable from v and u reachable from k. We can thus conclude that the minimal graph H will not have an edge from v to u. On the other hand if there is no k such that we can reach k from v and then u from k, then there must be an edge from v to u in H, because otherwise there would be no other way to get from v to u.

The algorithm to find H is therefore very simple. We iterate over all triples of vertices (k, u, v) and check if k is reachable from v and u reachable from k. If that is the case, we mark the edge from v to u as *not needed*. In the end H contains all edges that were not marked as not needed.

The time complexity is $O(n^3)$, where the slowest part is finding H.

Problem A: Adjusting passwords

Another IPSC has just started and you are late! You rush to your PC only to discover that you locked the screen and now you have to enter your password quickly to unlock it.

You are presented with a password prompt. It only supports the following keys:

Key	Action
a ... z	Enters the character.
enter	Submits the password.
backspace	Erases the last entered character, if any.

If you submit an invalid password, you will see an error message and a new, empty prompt will open.

Your password is P. In all the rush, you just typed the string Q into the prompt. It is possible that Q is not P: there may be a typo or two, or it can even be a completely different string.

Problem specification

Given P and Q, log in using as few additional keystrokes as possible.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. (In the easy subproblem A1 we have $t = 10$, in the hard subproblem A2 we have $t = 1000$.) Each test case is preceded by a blank line.

Each test case consists of two lines. The first line contains the correct password P and the second line contains the already typed string Q. Both are non-empty and have at most 50 characters.

Output specification

For each test case, output a line containing the list of keystrokes you want to press. Pressing enter is represented by `*` and pressing backspace is represented by `<`.

If there are multiple optimal solutions, you may output any of them.

Example

input

```
3

superfastawesome
superfastaxesome

superfastawesome
xuper

superfastawesome
superfastawe
```

output

```
<<<<<<wesome*
*superfastawesome*
some*
```

In the first test case, we keep pressing backspace until we delete the typo. In the second test case, it's faster to press enter immediately, receive an error message and begin anew from an empty prompt.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Monika Steinová
Quality assurance:	Michal 'Mimino' Danilák

Solution

Clearly, there are only two candidates for the optimal solution:

- Press Enter to clear the current password, type the correct password, and press Enter again.
- Press Backspace zero or more times to erase some characters from the end, then type the correct missing characters and press Enter.

We can easily count the length of the first candidate solution: it is the length of P, plus 2.

For the second candidate solution, we have one more question to answer: how many times should we press Backspace? The answer is again obvious:

- If Q is not a prefix of P, we have to press Backspace at least once.
- If Q is a prefix of P, it never pays off to press Backspace: if you erased a letter, later you would have to type in the same letter you just erased.

Thus, the second candidate solution can be implemented as follows: While Q is not a prefix of P, press backspace. Then, type the missing characters and press Enter.

In our program, we always construct both candidate solutions, compare their lengths, and output the shorter of the two.

Note that in the easy subproblem A1 we intentionally omitted some of the harder test cases. More precisely, in the test cases of A1 the string Q always contained at least one typo which was located in the prefix of P.

For extra credit, try giving a formal proof that one of the two candidates presented above must always be optimal. Think along the following lines: Can there be an optimal solution where I press Enter three or more times? Can there be one where I press Enter twice but press some other keys before the first Enter?

Problem B: Beating the game

You have probably heard about or played the game "2048". (No! Don't click the link now, bookmark it and go there after the contest!) The game is usually played on a 4 × 4 grid. Some cells of the grid contain tiles that have positive powers of 2 written on them. When the player chooses a direction, the tiles shift in that direction. If two equal tiles collide, they merge into a tile with their sum (the next power of 2).

There are also many derivatives of this game, with Fibonacci numbers, subatomic particles, or even Doge and Flappy Bird. Your friends are all raving about a new version that is played on a one-dimensional strip. This version is described below.

The game is played on an $1 \times n$ strip of cells. As usual, some of its cells are empty and some contain tiles with positive powers of two. Each move consists of three parts:

- First, the player chooses a direction – either left or right.
- Next, all tiles move in the chosen direction as far as they can. If a tile collides with another tile that has the same number, they merge together.
- If at least one tile moved or merged with another one in the previous step, a new tile randomly appears on one of the currently empty cells. The number on the new tile is usually a 2, but occasionally a 4.

The game ends when there is no move that would change the game state.

Moves and merges

When the player chooses a direction for tile movement, all tiles move in that direction one after another, starting with the tile that is closest to the target boundary. Whenever a tile collides into another tile that also has the same integer, the two tiles merge into one tile that has the sum of their numbers. The merged tile will occupy the cell that is closer to the target boundary, and cannot merge again in the current turn. All further collisions with the new tile are ignored.

Below are some examples to clarify these rules.

In our first example, the left diagram shows the original state and the right diagram the state after a move to the left (after all collisions are resolved and before the new random tile appears). Note that the two 4s did not merge into an 8.

```
+---+---+---+---+---+---+---+---+      +---+---+---+---+---+---+---+---+
| 2 |   | 2 | 2 | 2 |   |   | 2 |      | 4 | 4 | 2 |   |   |   |   |   |
+---+---+---+---+---+---+---+---+      +---+---+---+---+---+---+---+---+
```

Our second example shows a move to the right from the same initial situation. Note that different pairs of tiles merged this time – the ones closer to the right boundary.

```
+---+---+---+---+---+---+---+---+      +---+---+---+---+---+---+---+---+
| 2 |   | 2 | 2 | 2 |   |   | 2 |      |   |   |   |   |   | 2 | 4 | 4 |
+---+---+---+---+---+---+---+---+      +---+---+---+---+---+---+---+---+
```

For our final example, consider the situation shown in the left diagram below. In this situation, you can still move to the right – even though there is no space for the tiles to move, two of them can merge.

The result of a move to the right is shown in the right diagram below. Note that the two 8s did not merge, as one of them was created by a merge in this move. Also note that the 2 did shift to the right after the two tiles with 4 merged into an 8.

```
+---+---+---+---+---+---+---+---+      +---+---+---+---+---+---+---+---+
|   |   |   |   | 2 | 4 | 4 | 8 |      |   |   |   |   |   | 2 | 8 | 8 |
+---+---+---+---+---+---+---+---+      +---+---+---+---+---+---+---+---+
```

The player scores points for merging tiles. Merging two tiles with values $x/2$ into a new tile with value x is worth x points.

Pseudorandom generator

If at least one tile moved or merged with another one, a new tile randomly appears on one of the currently empty cells. Let `num_empty` be the number of currently empty cells. (Since at least one tile moved or merged, it must be at least 1.) The new tile's position and value is chosen with the following pseudocode:

```
pos = random() % num_empty

if (random() % 10) == 0:
    new_value = 4
else:
    new_value = 2

add_new_tile(cell = currently_empty_cells[pos], value = new_value)
```

In the pseudocode, `currently_empty_cells` is a list of all cells that are currently empty, from left to right. For example, if `pos` is 0 and `new_value` is 2, a new tile with the number 2 is created on the *leftmost* currently empty cell. The `%` symbol is the modulo operator.

To generate the values returned by `random()`, the game uses a deterministic pseudorandom generator: the subtraction with carry generator with values $(r, s, m) = (43, 22, 2^{32})$.

You know the initial seed of the pseudorandom generator: values x_0 through x_{42}. Each of the following values is generated using this formula:

$$\forall i \geq 43: \ x_i = (x_{i-s} - x_{i-r} - c(i-1)) \bmod m$$

In the above formula, "mod" is the mathematical modulo operator that always returns a value between 0 and $m-1$, inclusive. The function $c(i)$ is equal to 1 if $i \geq 43$ and $x_{i-s} - x_{i-r} - c(i-1) < 0$, and 0 otherwise.

The values $x_{43}, x_{44}, \ldots$ are the output of the pseudorandom generator. That is, the first call to `random()` returns the value x_{43}, and so on.

(You can check your implementation with this: If you set $x_0, x_1, \ldots, x_{42}$ to be $x_i = (999999999\, i^3) \bmod m$, the next three generated values should be 1050500563, 4071029865 and 4242540160.)

Problem specification

You decided to use your knowledge of the pseudorandom generator to cheat and defeat your friends by getting the highest score you possibly can.

In the easy subproblem B1, you are given the current state of a rather long strip, the values x_0 through x_{42}, and a sequence of attempted moves. Your task is to simulate the moves and output the game's final state.

In the hard subproblem B2, you are again given the values x_0 through x_{42} and the initial value of a short strip. The strip is empty except for one tile that has the number 2 or 4. Output the best possible score you can achieve.

Input specification

The first line of the input file contains an integer $t \leq 50$ specifying the number of test cases. Each test case is preceded by a blank line.

The first line of each test case contains an integer n, the length of the strip. For the easy subproblem B1 we have $2 \leq n \leq 1000$, for the hard subproblem B2 we have $2 \leq n \leq 9$.

The second line contains n integers, each of them either 0 (denoting an empty cell) or a power of two between 2 and 2^{62}, inclusive (denoting a tile with that number). These describe the current state from the left to the right. In the hard subproblem B2, $n-1$ of these integers are 0 and the remaining one is either 2 or 4.

The third line contains 43 integers: $x_0, x_1, \ldots, x_{42}$. Each of them is between 0 and $m-1$ (inclusive) and we chose them randomly.

This concludes the test case in the hard subproblem B2. In the easy subproblem B1, two more lines follow. The first line contains an integer a ($1 \leq a \leq 5000$): the number of attempted moves. The second line contains a string of a characters. Each of these characters is either '`l`' (representing a move to the left) or '`r`' (move to the right).

Output specification

Easy subproblem B1: For each test case, output a single line with n space-separated numbers – the state of the strip after performing all the moves, in the same format as in the input. (Note that some of the numbers in the output may be rather large.)

Hard subproblem B2: For each test case, output a single line with one number – the maximum score you can achieve for the given initial conditions.

Example for the easy subproblem B1

input

```
3

2
2 4
1 2 3 4 ... 42 43
2
rl

5
2 2 4 8 0
1 2 3 4 ... 42 43
3
lrl

5
2 2 4 8 0
43 42 41 40 ... 2 1
3
lrl
```

output

```
2 4
2 16 2 0 2
2 16 2 2 0
```

The "..." symbol is just to save space. The input file actually contains all 43 numbers.

Example for the hard subproblem B2

input

```
2

4
0 0 4 0
1 2 3 4 ... 42 43

3
2 0 0
43 42 41 40 ... 2 1
```

output

```
136
20
```

Task authors

Problemsetter:	Peter 'Bob' Fulla
Task preparation:	Jaroslav 'Sameth' Petrucha
Quality assurance:	Tomáš 'Tomi' Belan

Solution

For the easy task, all you had to do was to implement all the rules listed in the problem statement. Because the numbers in the output may exceed the size of a 64-bit integer, it was better to do everything with their binary logarithms. You could hard-wire the necessary powers of 2 as constants, use a language with arbitrary-precision integers, or resort to dirty tricks: large powers of 2 can be represented exactly in a `double`, and a `printf("%.0f",x)` prints them correctly.

Now that we have the simulation out of our way, we can focus on the hard task. First feasible idea: Let's solve it by brute force! At each moment, there are at most two ways to make a move: to the left, or to the right. After finitely many moves, our strip will be filled with numbers and no valid move will be possible. (With a strip of length n, at any point in the game, there cannot be two tiles with the number 2^{n+1}.)

This is almost sufficient for solving the task. The last thing we have to notice is that there are only few possible states of the strip, and we encounter each many times. This is because the numbers cannot be in arbitrary order. (Do you see why? Can you make a more precise statement about how the reachable states look like?)

Therefore, if we improve the brute force program with memoization, the program will run fast enough. (The key for memoization should be both the state of the strip and the number of steps. This is because due to the 2 vs. 4 being added, it is possible to reach the same state in different numbers of steps, which leads to different random choices in the future.)

Problem C: Copier

We have a strange box with a big red button. There is a sequence of integers in the box. Whenever we push the big red button, the sequence in the box changes. We call the box a "copier", because the new sequence is created from the old one by copying some contiguous section.

More precisely, each time the red button is pushed the copier does the following: Suppose that the current sequence in the box is $a_0, a_1, a_2, \dots, a_{m-1}$. The copier chooses some i, j, k such that $0 \le i < j \le k \le m$. Then the copier inserts a copy of $a_i, \dots, a_{j-1}$ immediately after a_{k-1}. Note that $j \le k$: the copy is always inserted to the right of the original. Here is how the sequence looks like after the insertion:

$$a_0, \dots, a_{i-1}, \underbrace{a_i, \dots, a_{j-1}}_{\text{original}}, a_j, \dots, a_{k-1}, \underbrace{a_i, \dots, a_{j-1}}_{\text{copy}}, a_k, \dots, a_{m-1}$$

Problem specification

In the morning we had some **permutation** of $1 \dots \ell$ in the box. Then we pushed the button zero or more times. Each time we pushed the button, a new (not necessarily different) triple (i, j, k) was chosen and the sequence was modified as described above. You are given the sequence S that was in the copier at the end of the day. Reconstruct the original permutation.

Input specification

The first line of the input file contains an integer $t \le 60$ 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 an integer n ($3 \le n \le 100\,000$) – the length of the final sequence S. The second line contains n integers – the sequence S. For each test case, there is a positive integer ℓ such that S can be produced from some permutation of $\{1, 2, \dots, \ell\}$ using a finite sequence of copier operations.

In the **easy subproblem C1** you may also assume that $n \le 10\,000$, that the sequence S was obtained from some permutation by pushing the red button exactly once, and that the copier chose $j = k$, i.e., it inserted the copied subsequence immediately after the original.

Output specification

For each test case, output a single line with a space-separated list of integers: the original permutation. If there are multiple valid solutions, output any of them.

Example

input

```
3

7
5 1 4 1 4 2 3

11
4 3 1 2 3 1 4 3 3 1 4

7
1 1 1 1 1 1 1
```

output

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

The first test case satisfies the conditions for the easy subproblem, the copier duplicated the subsequence 1 4. In the second test case we started with 4 3 1 2, changed it into (4 3) 1 2 (4 3), then changed that sequence into 4 (3 1) 2 (3 1) 4 3, and finally changed that into 4 3 1 2 (3 1 4) 3 (3 1 4).

Task authors

Problemsetter:	Ján 'JaNo' Hozza
Task preparation:	Ján 'JaNo' Hozza
Quality assurance:	Peter 'Bob' Fulla

Solution

The important observation is that the copy is always placed to the right of the original. Hence, the copier operation does not change the order in which numbers appear in our sequence for the first time. Therefore we may just iterate over the sequence and filter out the elements we have already seen.

Problem D: Disk space eater

“We could always ask them to find a message in a file full of zeros.”
“Isn’t that too easy?”
“Then let’s make the file huge, like 1 EB. That would be hard, right?”
“Sure it would, because nobody can download an exabyte of data.”
“We can compress it! And if it is still too large, we can compress it again and again. And again!”

Compression algorithm

The input files are compressed with bzip2. Bzip2 is a block-based compression algorithm that takes a single file and produces a compressed version of the same file. For more information on bzip2, see its Wikipedia entry or the project web page.

If you are on Linux or a Mac, you probably already have bzip2 installed (just type bzip2 or bunzip2 into your terminal). If not, install the `bzip2` package. There is also bzip2 for Windows.

Problem specification

In each subproblem you are given a file that has been compressed using bzip2 one or more times. Your task is to extract a message from the original file. The original file contains exactly one message: a contiguous block of printable non-whitespace ASCII characters. The message has at most 100 characters. Every other byte in the original file is a zero byte.

In the easy subproblem, the size of the original file is roughly 1 TB and it was compressed twice in a row. The hard subproblem has four nested layers. Have fun!

Output specification

Submit a text file with the extracted message.

Task authors

Problemsetter:	Michal 'Mic' Nánási
Task preparation:	Michal 'Mic' Nánási
Quality assurance:	Peter 'Ppershing' Perešíni

Solution

The easy subproblem was actually one of the easiest problems ever. All you had to do was to follow the instructions: unzip the file twice and remove bytes with value zero. In a unix-like environment this can be done as follows:

```
cat easy.in | bunzip2 | bunzip2 | tr -d '\000'
```

(Or maybe use `strings` instead of the `tr`.)

If we processed the uncompressed stream at speed around 175MB/s, it would take us around 100 minutes, which is about 1/3 of the contest length. (Most modern machines should be faster than that.)

In the hard subproblem we might be tempted to do the same thing:

```
cat hard.in | bunzip2 | bunzip2 | bunzip2 | bunzip2 | tr -d '\000'
```

However, this would take about 349525.3 contests (almost 8 years). Unless you have a time machine, you are not going to solve the problem this way. In case you wonder, the uncompressed file for the hard input does indeed have 1 EB. (And since compression is slower than decompression, it would take much more than 8 years to create the hard input using only bzip2.)

How to do it faster? We need to understand the format of bzip2 better. In a nutshell, bzip2 compresses the input stream block by block. Block sizes in both input and output stream vary, but if two input blocks are same, then also compressed blocks are same and vice versa. The maximum size of uncompressed data in the compressed block is around 46 MB. Each block starts with the *compressed magic* string (`1AY&SY`, or `31 41 59 26 53 59` in hexadecimal values). This should be enough information to solve the task.

The general idea is to discard blocks that encode only zeros. We will end up with a much smaller file, which can then be decompressed. We will apply this to two layers of compression (first and second compression).

At first, we bunzip the easy input once, to see the original file compressed with one layer of compression. Then we look at the hexdump of resulting archive:

```
mic@ipsc$ cat easy.in | bunzip2 > easy.1.bz2
mic@ipsc$ xxd easy.1.bz2
0000000: 425a 6839 3141 5926 5359 0e09 e2df 015f  BZh91AY&SY....._
0000010: 8e40 00c0 0000 0820 0030 804d 4642 a025  .@..... .0.MFB.%
0000020: a90a 8097 3141 5926 5359 0e09 e2df 015f  ....1AY&SY....._
0000030: 8e40 00c0 0000 0820 0030 804d 4642 a025  .@..... .0.MFB.%
0000040: a90a 8097 3141 5926 5359 0e09 e2df 015f  ....1AY&SY....._
0000050: 8e40 00c0 0000 0820 0030 804d 4642 a025  .@..... .0.MFB.%
0000060: a90a 8097 3141 5926 5359 0e09 e2df 015f  ....1AY&SY....._
...
0046360: a90a 8097 3141 5926 5359 0e09 e2df 015f  ....1AY&SY....._
0046370: 8e40 00c0 0000 0820 0030 804d 4642 a025  .@..... .0.MFB.%
0046380: a90a 8097 3141 5926 5359 fa81 7094 0012  ....1AY&SY..p...
```

```
0046390: 03c7 20c0 0000 04c0 000b 2000 0100 0820  .. ....... ....
00463a0: 0050 8018 09aa 881b 5324 5404 bdda a490  .P......S$T.....
00463b0: 555f 6f35 99ae 7338 4540 4bf3 1415 9265  U_o5..s8E@K....e
00463c0: 3590 e09e 2df0 15f8 e400 0c00 0000 8200  5...-...........
00463d0: 0308 04d4 642a 025a 90a8 0973 1415 9265  ....d*.Z...s...e
00463e0: 3590 e09e 2df0 15f8 e400 0c00 0000 8200  5...-...........
00463f0: 0308 04d4 642a 025a 90a8 0973 1415 9265  ....d*.Z...s...e
...
00c43c0: 3590 e09e 2df0 15f8 e400 0c00 0000 8200  5...-...........
00c43d0: 0308 04d4 642a 025a 90a8 0973 1415 9265  ....d*.Z...s...e
00c43e0: 359f 0f47 1b50 0bbc f400 8c00 0002 0000  5..G.P..........
00c43f0: 8200 030c c09a a69a 0a08 5b50 2821 78bb  ..........[P(!x.
00c4400: 9229 c284 855e d959                      .)...^.Y
```

As we see, there is the *compressed magic* string every 32 bytes indicating, that the compressed file is composed from 32 byte long blocks (let's call them *level 1* blocks), each containing some number of compressed zeros (in fact, each block compress around 46MB of zeros). Then there is a change of the pattern indicating the change in the input file, the selected message. And after this special part of the bzipped file, it's back to the 32-byte pattern. This pattern then breaks sligthly before the end of the file – the last block is different, because it encodes a different number of zeros). The second repetitive pattern encodes the same number of zeros as the first repetitive pattern, but it looks different. If we look closer on the hex values, we see that it is in fact the same as the first pattern, but shifted by 4 bits. This is because the length of the block in bzip2 is not aligned to a full byte; search for the π in the hex dump (in case you didn't notice, *compressed magic* in hex is the first 12 digits of π).

Since the message is short, it is encoded in at most two blocks. (It would fit into one, but it is possible that it lies on the boundary of two blocks.) Either way, the repetitive blocks can be removed, because they encode only zeros. The resulting archive would contain the message with much fewer zeros.

Note that when we remove a block from a bzip file, the checksum in the end of the file will be incorrect and bunzip2 will refuse to decompress it. But when you provide the compressed file through standard input, bunzip2 will decompress the file and write it to the standard output ***before*** *it checks the checksum and quits with error.*

For the easy subproblem, we can use the above observation to get the solution in a matter of seconds: First, we unpack the input file once, then we discard the repetitive level 1 blocks, and then we unpack the resulting archive a second time.

For the hard subproblem, probably the best course of actions was the following one: First, take the input file and unpack it twice to get a twice-compressed archive. The size of this archive is approximately 140 MB. Now, in this archive most of the blocks (*level 2* blocks) will again look the same. We should remove most of them somehow. But can we do so without damaging the once-compressed file?

The problem is that the level 2 blocks don't necessarily contain several level 1 blocks each. It is possible that a level 2 block both starts and ends within some level 1 block. Luckily for us, whenever we find multiple identical level 2 blocks that immediately follow one another, we can conclude that they must have the same offset within the level 1 blocks, hence we can discard them without damaging the underlying archive.

To sum this up, the algorithm for the hard input is following (easy input skips the first step):

1. Decompress input file twice using bunzip2
2. Remove all repetitive level 2 blocks from the compressed file
3. Decompress the trimmed file
4. Remove all repetitive level 1 blocks from the compressed file

5. Decompress the trimmed file

Implementing all those steps is more or less straightforward, the only thing you should watch out for is that the blocks are not aligned to full bytes, therefore searching for *compressed magic* by bytes will not find all blocks. It is possible to generate all shifts of the *compressed magic*, remove the flanking bits and search for those strings.

Problem E: Empathy system

The Institute for Promoting Social Change is building the world's first Artificial Intelligence. To make sure the AI won't decide to eradicate humanity, it will be programmed to make humans happy. But first, the AI must be able to actually *recognize* human emotions. We probably don't want an AI which thinks that humans are happy when they are screaming and frantically running around. You have been tasked with fixing this deficiency by programming the AI's empathy system.

Problem specification

Your task is simple: Write a program that will process a photo of a human being and determine whether the person is *happy*, *sad*, *pensive*, or *surprised*.

This problem is special. Usually, you get all input data, you can write a program in any programming language, and you only submit the computed output data, not your source code. This task is the exact opposite. You *only* submit source code – specifically, a program written in the Lua 5.2 language. (See below for an introduction.) We will run this program on our test cases and tell you the results.

Your program won't have OS access, so input and output will happen through global variables. The input is a 128 × 128 photograph. Each pixel has three color channels (red, green, and blue) stored as integers between 0 and 255, inclusive. The photo will be in a 128 × 128 × 3 array named `image`. Lua arrays are indexed from 1, so for example, the blue value of the bottom left pixel will be in `image[128][1][3]`.

After recognizing the human's emotion, your program must set the variable `answer` to the number `1` if it's happy, `2` if it's sad, `3` if it's pensive and `4` if it's surprised.

Limits

The libraries and functions named `debug.debug`, `io`, `math.random`, `math.randomseed`, `os`, `package`, `print` and `require` have been removed. Other functions that try to read files will fail.

Our data set consists of 200 photographs. We will sequentially run your program on every photo. Every execution must use under 512 MB of memory, and the whole process must take less than 15 seconds. If your program exceeds these limits, or causes a syntax or runtime error, you will receive a message detailing what happened.

If the program produces an answer on all 200 images without causing any errors, we will then check which answers are correct, and tell you the detailed results. The program doesn't have to answer everything correctly to be accepted. In the **easy subproblem E1**, the number of correct answers must be at least 60, while in the **hard subproblem E2**, it must be at least 190. (In other words, the accuracy of your program must be at least 30% for the easy subproblem, and at least 95% for the hard subproblem.)

Both subproblems use the same image set. The only difference is in the acceptance threshold.

You can make 20 submissions per subproblem instead of the usual 10.

Testing your program

We have given you the first 12 of the 200 photos that comprise our data set. The other photos are similar, so you can use them to check if your program is working correctly. If you have Lua installed and your program is named `myprogram.lua`, you can test it with this command:

```
lua -l image001 -l myprogram -e "print(answer)"
```

If you can't install Lua, you can use the on-line interpreter at repl.it. Copy and paste the contents of `image001.lua` and `myprogram.lua` to the editor on the left, then add `print(answer)`, and press the Run button in the middle. Note that repl.it is very slow (it actually compiles the Lua virtual machine from C to JavaScript), and it uses Lua version 5.1 instead of 5.2. But if your program works in repl.it, it will likely also work on our servers.

Introduction to Lua

If you don't know the Lua language, here's a quick introduction:

- Variables: `foo_bar = a + b`
- Literals: `nil`, `true`, `false`, `123`, `0xFF`, `"a string"`
- Arithmetic: `1+2 == 3`, `1-2 == -1`, `2*3 == 6`, `1/2 == 0.5`, `6%5 == 1`, `2^3 == 8`
- Comparison: `a == b`, `a ~= b`, `a < b`, `a <= b`, `a > b`, `a >= b`
- Logic: `a and b`, `a or b`, `not a` (note that `nil` and `false` are considered false, while everything else is considered true – including `0` and `""`)
- Math: `math.abs(x)`, `math.floor(x)`, `math.max(a, b, c, d)`, `math.pi`, etc.
- Binary: `bit32.band(a, b)`, `bit32.bor(a, b)`, `bit32.bnot(a)`, `bit32.bxor(a, b)`, etc.
- "If" statement: `if cond then ... elseif cond then ... else ... end`
- "While" loop: `while cond do ... end`
- Numeric "for" loop: `for i = 1, 128 do ... end`
- Functions: `function some_name(a, b, c) local d = 7; return a + d; end`
- Local variables (in functions and control structures): `local l = 123`
- Arrays (tables): `arr[idx]` (indexed from 1), `{ 1, 2, 3 }` (new array), `#arr` (array length)
- Comments: `--[[ multiple lines ]]`, `-- until end of line`
- Newlines and semicolons are optional: `a = 1 b = 2 + 3 c = 4 * 5`

For more details, refer to the Lua manual and the language grammar, or other on-line resources – though you probably won't need most parts.

Example

Your submission could look like this:

```
num_bright_pixels = 0
for y = 1, 128 do
  for x = 1, 128 do
    if image[y][x][1] + image[y][x][2] + image[y][x][3] > 600 then
      num_bright_pixels = num_bright_pixels + 1
    end
  end
end

if num_bright_pixels > 2000 then
  answer = 1   -- bright photos are happy
else
  answer = 2   -- dark photos are sad
end
```

This is a valid program that produces an answer for each image. But the answers aren't correct, so you'd receive this response:

```
Wrong answer: Only 47 answers are correct.
1 is right, 2 is right, 2 is wrong, 1 is wrong, 1 is wrong, ...
```

The program correctly identified that the first person (`image001`) is happy and the second person (`image002`) is sad, but it can't tell that the third person is happy, the fourth person is surprised, etc.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Tomáš 'Tomi' Belan
Quality assurance:	Michal 'mišof' Forišek

Solution

Photo credits, in alphabetical order: Tomáš Belan, Ladislav Bačo, Askar Gafurov, Marián Horňák, Michal Nanási, Rastislav Rabatin, Kamila Součková, and last but not least our special guest star: *Darth Vader*.

Background

The idea for this problem was created on the IOI International Scientific Committee meeting before IOI 2013. Some of you might recall the Art Class problem from this contest. The problem looked similar – you have a bunch of images, classify them into four groups. The differences: you got more data for testing, and the four groups were much easier to separate from each other. During said meeting we realized that we had to restrict the "full feedback" contestants have during the contest, otherwise somebody may be able to game the system. Hence, IOI 2013 contestants did only get aggregate feedback for their submissions... and now in IPSC 2014 here is the problem where you *have to* game the system.

It's clearly infeasible to make a real multiclass classifier. You have no number crunching libraries, the time limit is unforgiving, the training set is minuscule, and the accuracy requirement is absurdly high. If you want to solve the problem, you must resort to trickery.

Easy subproblem

Getting close to 30% accuracy is not that difficult. After all, a program that answers randomly has a 25% chance of hitting the correct answer in each test case. Your program can't use `math.random()`, but that doesn't matter – just use a simple hash function that adds or xors together a few pixels of the image. It's a noisy photo, so this is enough to produce nearly uniformly distributed answers.

The expected accuracy of programs like this is 25%. You could try different hash functions until you find a good one, but the likelihood of reaching 30% accuracy (60 correct answers) is low. Can you use some additional piece of information to improve the program's accuracy?

Yes, you can. You have the first 12 of the 200 images, so you already know both the first 12 inputs and their correct answers. And that means you can hardcode them. The program will look at a few pixels of `image`, check if they match one of the 12 known images, and in that case, just output the corresponding answer.

Let's look at the expected number of correct answers. Before this improvement, a random program would only solve 3 of the first 12 images. Now that it can solve all 12, the expected total raises from 50 to 59. That's almost 60, so the probability of success is almost 50%. The easy subproblem can be solved in very few submissions, often just a single one, by trying different hash functions.

Alternately, if your random submission got *very few* answers right, it is a good idea to use the same hash function but shift the answer cyclically.

Of course, the best solution was actually to use multiple submissions on E1. This will be explained below.

Hard subproblem

The hard subproblem requires almost 100% accuracy, so our previous tricks won't be enough. But the grader has a big weakness: It gives you full feedback. You get all the answers your program computed, and learn which ones were correct. This can give you a lot of information.

For example, you could submit the one-liner "`answer = 1`" and learn which test cases have 1 as the correct answer. In three submissions, you could learn all the correct answers. If your program knew the test case number, you could just hardcode all the answers and you'd be done. But it doesn't, so you need a way to tell the test cases apart using only the pixel data.

This can be achieved by using more hash functions. When you submit a program that computes the value of a hash function, you learn the hash function's value for each of the 200 images. If you do that enough times, all images will be pairwise different in at least one of your hash functions, and so, you'll be able to distinguish between them.

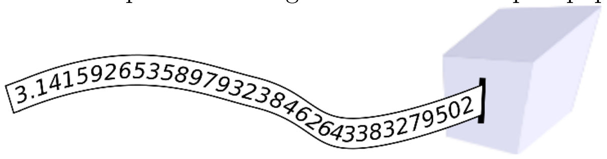
Your final program will simply compute all your hash functions, use the results to recognize which test case is currently running, and output the appropriate answer.

Note that *Wrong answers* in easy subproblems are twice as cheap as in hard subproblems. This has an interesting effect: Solving the hard subproblem requires many *Wrong answers*, but you can make them in either subproblem! The best strategy was therefore to make enough bad submissions on E1, and then solve both subproblems, with just a single submission on E2.

(Of course, it can happen that you solve E1 by accident before you have enough information to solve E2. Thus, the ultimate trick is: while you don't want to solve E1, you should give *wrong answers* for the first 12 images on purpose. This decreases the probability of accidentally solving E1. Neat, huh?)

Problem F: Find a sudoku in π

Mikey once built a machine that printed the digits of π onto a strip of paper:



Cut out some segments of the paper strip and assemble them into a valid sudoku square.

Pi and indexing into its decimal expansion

People often amuse themselves by looking for some specific sequences of decimal digits in π. For example, Mikey was born on 1992-06-05, so he was pleased to note that the sequence "9265" does indeed appear in π: its first occurrence starts with the fifth decimal digit of π. We say that "9265" occurs at *offset* 5.

It is often claimed that one can find everything somewhere in π, including the collected works of Shakespeare. However, this is actually still an open problem. Basically, we only know is that the short substrings seem to have a roughly-uniform distribution in the known prefix of π.

Mikey became interested in substrings of π that are permutations of digits 1 through 9. The first such substring is easy to find: "617539284" occurs at offset 2186. The next one follows immediatelly: "175392846" at offset 2187. (Note that these two substrings share 8 of their 9 digits.)

By the time Mikey printed the first $13,000,000,000$ digits of π, he already saw each of the 9! possible substrings at least once. (On average, he saw each of them roughly 13 times.) The last permutation of 1-9 to appear was "631279485". The first offset in π where you can find this particular substring is $12,944,969,369$.

Sudoku specification

A *sudoku square* is a 9×9 square of digits. Each row, each column, and each of the nine highlighted 3×3 blocks of a sudoku square must contain each of the digits 1 through 9 exactly once. One valid sudoku square is shown in the following figure.

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

Creating a sudoku square

Mikey wants to create a sudoku square using the following process: First, he will take scissors and cut out nine **non-overlapping** segments of his strip of paper. Each of the nine segments has to contain some 9-digit substring of π that is a permutation of digits 1 through 9. Then, Mikey will place those 9 segments one below another in any order he likes. (That is, each of the pieces of paper becomes one row of a 9×9 grid of digits.)

Problem specification

In each subproblem your goal is the same: you have to choose 9 substrings of π and arrange them into rows of a grid in such a way that the resulting grid becomes a valid sudoku square.

In the **easy subproblem F1** you can use the first $250,000,000$ decimal digits of π. Any valid solution will be accepted.

In the **hard subproblem F2** you must use a prefix of π that is *as short as possible*. That is, if we look at the offsets of the substrings you chose, the largest of those 9 offsets must be as small as possible. Any such solution will be accepted.

Output specification

The output file must contain exactly nine lines. Each line must have the form "string offset", where "string" is a permutation of 1 through 9 and "offset" is a valid offset into π where this particular string of digits occurs.

The strings must form a valid sudoku square, in the order in which they appear in your output file. The offsets must be small enough (as specified above).

Example output

Below is a syntactically correct example of an output file you might submit.

```
617539284 2186
175392846 2187
631279485 12944969369
123456789 523551502
123456789 523551502
123456789 523551502
123456789 523551502
123456789 523551502
123456789 773349079
```

Note that we would judge this particular submission as incorrect, for two reasons. First, even though all the given offsets are correct, some of them are too large, even for the easy subproblem. Second, that is not a valid sudoku square. (For example, the digit 6 appears twice in the top left 3×3 square.)

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Michal 'Žaba' Anderle

Solution

We will first show how to obtain π with the necessary precision, then we'll discuss solving the easy and the hard subproblem, respectively.

Getting pi

Pi has already been computed to an awful lot of decimal places. The computed digits can be downloaded from many corners of the internet, including `http://piworld.calico.jp/epivalue1.html` and `ftp://pi.super-computing.org/.1/pi200m/`.

Of course, you could also simply compute the first 250M digits on your own machine. One of the better programs that are available is the `gmp-chudnovsky.c` program: an implementation of the Chudnovsky algorithm (`https://en.wikipedia.org/wiki/Chudnovsky_algorithm`) by Hanhong Xue. The program is written in C using the GMP library and you can download it here: `https://gmplib.org/pi-with-gmp.html`

If you are interested in writing your own program to compute π or some other constant, `https://en.wikipedia.org/wiki/Spigot_algorithm` might be a good place where to start.

Parsing pi

Now that we have π to an awful lot of decimal places, let's parse it. More precisely, let's find all the length-9 substrings we are interested in. This can easily be done in linear time.

For the first 250M digits of π, we will find 90 123 unique permutations (some with more than one occurrence). We can simply store a list of these occurrences, and discard the rest of π. Dealing with a 90k-line file seems much more manageable than dealing with 250M characters, right?

Note that there are $9! = 362\,880$ different permutations of 1 through 9. As we have 90 123 of them in our prefix of π, this means that we can use approximately 1/4 of all permutations when constructing our sudoku grid in the easy subproblem.

Solving the easy subproblem

The estimate we just made is quite promising. Let's do a back-of-the-envelope estimate. Suppose that we just generate random sudoku grids, and each time we generate one we check whether its 9 rows represent valid permutations. The probability that all 9 will be valid is about $(1/4)^9 \approx 1/250\,000$. Thus, after generating about 250k random sudoku grids, we should expect one of them to be a valid solution to the easy subproblem.

That sounds perfectly reasonable, especially if we consider that by pruning early we can find a valid solution much sooner.

All we need is a way how to generate a bunch of random sudoku grids. How can we do that? Easily. Just write the standard backtracking algorithm that solves a given sudoku. Then, you randomize the choices it makes: if you cannot deduce anything new, pick a cell and fill it with a random value that is still consistent with your deductions. Finally, you just run this algorithm starting with an empty grid.

As additional pruning, you can verify the validity of rows during the algorithm: each time you add a new value, check whether there is a valid permutation that corresponds to your partially-filled row.

Solving the easy subproblem – another way

Instead of generating random sudoku grids and then checking whether their rows are valid, we can also use a different approach: one that matches more closely what the statement asks us to do.

We will use backtracking again, but this time we will simply try to build a valid sudoku grid row by row, using only the valid permutations we have. After we place a row, we update the set of missing digits in each column, and using those we then find all permutations that are still usable for the remaining rows. After we fill in 3, 6, and 9 rows completely, we run an additional check to verify that the 3×3 blocks are filled correctly.

A note on overlapping occurrences

So far we did not mention anything about handling the constraint that we must use nine *non-overlapping* occurrences of a permutation. How do we handle this constraint? By not handling it at all. The chance that we would find a solution that violates this constraint is negligibly small. Plus, we can always verify it by hand at the end, and if we happen to be really really unlucky, we can always run our program again and generate a different solution.

This is much faster and simpler than writing multiple lines of code that would handle this constraint. The same will be true for the hard subproblem: we will simply generate the best solution without checking this constraint, and then we'll be pleased to discover that the best solution has no overlapping permutations.

Solving the hard subproblem

The hard subproblem can also be solved, basically, by brute force. We just have to be smarter and prune our search better. Below we describe one method that worked well for us; we are certain that there are many other approaches that work just as well, if not better.

In the previous solution for the easy subproblem, many branches failed when we finished some 3×3 blocks and discovered that they are not filled correctly. Most triples of permutations, even if they are column-compatible (i.e., no two have the same number at the same position) will produce invalid 3×3 blocks.

Here are some exact numbers to make this more obvious: Consider the first 1000 distinct permutations of 1 through 9 that occur in π. We can form $\binom{1000}{3} = 166\,167\,000$ different triples out of them. Out of those, 7 249 826 are column-compatible, but only 3 164 of those also produces valid 3×3 blocks.

This leads us to the following algorithm: Read the permutations from the input in the order in which they first appear in π. Each time you read a new permutation, try pairing it up with two previously-read permutations to produce a valid triple of rows. And each time you find a new valid triple, try combining it with two previously-found triples to obtain a valid sudoku grid.

It should be obvious that this algorithm does indeed solve the hard subproblem – we never skip any possible solution, we just use clever tricks to prune away many branches that don't contain any valid solutions.

When executed, this algorithm finds a solution after processing the first 3143 distinct permutations that occur in π. Using those 3143 permutations, only 102 196 valid triples can be produced. Additionally, almost all those triples are incompatible: only 196 415 pairs go together. And once you have two triples, you know exactly what values the third triple of rows should have in each column – and any such triple will go with the other two to produce a valid sudoku grid.

Problem G: Game on a conveyor belt

Adam and Betka finally found a summer job: they were hired in a conveyor belt sushi restaurant. The work is hard and the pay is low, but one fantastic perk makes up for that: After closing time, they may eat all the sushi left on the belt! However, there are still the dishes to wash. As neither of them feels enthusiastic about this task, Betka suggested playing a game she just invented. The winner will be allowed to go home immediately, and the loser will have to stay and slog through the pile of dishes.

Problem specification

The game involves eating sushi from the conveyor belt. You can visualize the visible part of the belt as a sequence of trays moving from right to left. Some of the trays carry a plate with a piece of sushi, and all the remaining trays are empty. The belt moves at a constant rate: every second all trays are shifted by one position to the left. The leftmost tray disappears through a hatch into the kitchen, and a new empty tray arrives at the rightmost position. If there was a piece of sushi on the tray that went into the kitchen, a trainee cook gets rid of it.

Adam and Betka take alternate turns, starting with Adam. Each turn takes exactly one second. In each turn, the current player picks up one piece of sushi and eats it. While the player is eating the sushi, the belt rotates by one position. If a player cannot choose any piece of sushi because all the trays are empty, that player loses the game.

Given the initial contents of the belt, determine the winner of the game (assuming both players choose their moves optimally in order to win).

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 an initial state of the belt. The first line contains a positive integer n ($1 \leq n \leq 100\,000$) – the number of trays. The second line consists of n digits `0` and `1` representing the trays on the belt from the left to the right. (I.e., the first character represents the tray that will enter the kitchen first.) Empty trays correspond to zeros and trays with a plate of sushi to ones.

In each test case of the **easy subproblem G1**, there are at most 20 trays with a piece of sushi. Moreover, n does not exceed 100.

Output specification

For each test case, output one line with the winner's name (`Adam` or `Betka`).

Example

input

```
2

5
01010

6
110101
```

output

```
Betka
Adam
```

In the second test case, Adam will start by eating from the second tray. At the beginning of Betka's following turn, the state of the belt will be `001010`.

Task authors

Problemsetter:	Askar Gafurov
Task preparation:	Peter 'Bob' Fulla
Quality assurance:	Askar Gafurov

Solution

Betka's game on a conveyor belt is an impartial combinatorial game, hence every state of the belt is either winning or losing for the current player. If you are not familiar with combinatorial games, we recommend the book Game Theory by Thomas S. Ferguson for an introduction to the topic.

In this problem, we need to determine the category (i.e. winning or losing) to which a given initial state belongs. Let us note that we will avoid the conventional wording "winning/losing positions" throughout this analysis and use "winning/losing states" instead. The word "position" will refer solely to a location on the belt.

Easy subproblem

We can easily solve this subproblem using recursion: If all trays in the given state P are empty, the current player cannot move and the state is trivially losing. Otherwise, the current player may choose any of the non-empty trays and move by eating the sushi on that tray. This move leads to a state Q with fewer ones than P has, so we can recursively determine whether Q is winning or losing without getting into an infinite loop. If there is a move such that it transforms P into a losing state, then P itself is winning. However, if all the available moves lead to winning states, P is losing.

This solution works, although it is impractically slow. We can improve it by adding memoization: When we determine the category of a state P for the first time, we store P together with the result in a `map` or a hash table. The next time we are interested in the category of P, we can skip the computation and return the remembered result immediately. In other words, we carry out the computation for any state at most once.

We will show that the number of states that are reachable from the initial state is reasonably low. Let us denote the number of ones in the initial state of the belt by m. Clearly, the game cannot last longer than for m moves. After i moves, there are at most m positions on the belt where a non-empty tray may occur, because any non-empty tray was shifted to its place from a position containing `1` in the initial state. Some of these m positions may be empty, so there are at most 2^m reachable states of the belt after i moves of the game. Overall, the number of reachable states does not exceed $m \cdot 2^m < 21 \cdot 10^6$.

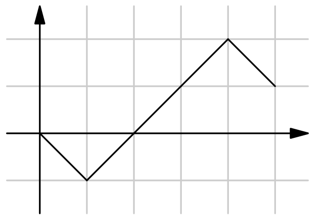
Hard subproblem

The number of reachable states grows exponentially, therefore we cannot hope to examine all of them in the hard subproblem. Fortunately, the winning/losing states exhibit an intricate pattern. To solve the hard subproblem, one just needs to discover that pattern and implement an algorithm checking for it.

Once again, we will denote the number of non-empty trays in the initial state by m. The circular nature of the belt is irrelevant for the game, as only empty trays appear at the rightmost position. Hence, we can think of the belt as a one-way infinite string – we just take the string representation of the visible part of the belt and append infinitely many zeros after its right end.

Let p be a finite string consisting of zeros and ones. We define zeros(p) as the number of zeros in p, ones(p) as the number of ones in p, and difference(p) as ones(p) − zeros(p). Let us define peak(p) as the maximum of difference(q) for all strings q such that q is a prefix of p. For example, zeros(`01110`) = 2, ones(`01110`) $= 3$, and therefore difference(`01110`) $= 1$. But peak(`01110`) $= 2$, because difference(`0111`) $= 2$ and `0111` is a prefix of `01110`.

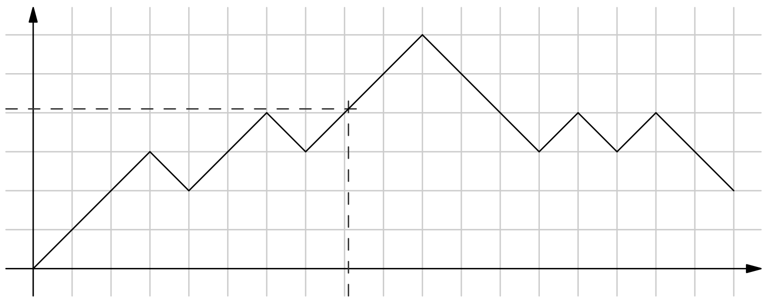
A simple way how to visualize the peak of a string is to interpret every zero of the string as -1, every one as $+1$, and plot the progression of prefix sums of the obtained sequence. The peak is then equal to the maximum of these sums. On the following figure, you can see such a visualization for string `01110`.



Before describing the pattern, let us define one last auxiliary function: $g(m, \ell) = \frac{2m-3\ell}{4}$, where m is the number of ones in the initial state and ℓ is an integer. We are only interested in cases when $g(m, \ell)$ is also an integer. This restricts the set of possible values of ℓ: If m is even, ℓ must be divisible by 4. If m is odd, ℓ must give remainder 2 when divided by 4.

Finally, we present a characterization of losing states: A state s is losing iff there is an integer ℓ and a prefix p of string s such that zeros$(p) = g(m, \ell)$ and peak$(p) = \ell$.

For example, state `111011011100010100...` is losing, because its prefix `11101101` has the peak equal to 4 and contains $g(10, 4) = 2$ zeros.



How to determine whether a given state satisfies this condition? Firstly, we do not know the right value of ℓ, but there is only a finite number of reasonable integers to try: peak(p) and zeros(p) are non-negative for any string p, therefore it must hold $0 \leq \ell \leq \frac{2}{3}m$. When we choose a value of ℓ, we can simply check whether any of the prefixes with $g(m, \ell)$ zeros has the peak equal to ℓ. Implementing this approach properly, one obtains an algorithm with time complexity $O(n)$.

Problem H: Hashsets

There are not many data structures that are used in practice more frequently than hashsets and hashmaps (also known as associative arrays). They get a lot of praise, and deserve most of it. However, people often overestimate their capabilities. The Internet is full of bad advice such as "just use a hashset, all operations are O(1)" and "don't worry, it always works in practice". We hope that you know better. And if you don't, now you'll learn.

Let's start by looking at a sample program in two of the most common modern programming languages: C++11 and Java 7.

```
// C++11
#include <iostream>
#include <unordered_set>
int main() {
    long long tmp;
    std::unordered_set<long long> hashset;
    while (std::cin >> tmp) hashset.insert(tmp);
}

// Java
import java.io.*;
import java.util.*;
public class HashSetDemo {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader( new InputStreamReader( System.in ) );
        HashSet<Long> hashset = new HashSet<Long>();
        for (String x = in.readLine(); x != null ; x = in.readLine())
            hashset.add( Long.parseLong(x) );
    }
}
```

Both programs do the same thing: they read a sequence of signed 64-bit values from the standard input and they insert them into a hashset.

We compiled and ran both programs on our server. If we used the sequence $1, 2, \ldots, 50\,000$ as the input, the C++ program needed about 0.05 seconds and the Java program needed about 0.25 seconds. After we increased the input to $1, 2, \ldots, 1\,000\,000$, the C++ program ran in 0.6 seconds and the Java program took 0.8 seconds. That's fast, right?

Based on this evidence and their limited understanding, many people would claim that the above programs will process any sequence of n integers in $O(n)$ time. Are you one of those people?

Problem specification

Submit a sequence of 64-bit integers. The length of your sequence can be at most **50 000**.

To solve the **easy subproblem H1**, your sequence must force **at least one** of our sample programs to run for at least 2 seconds on our server.

To solve the **hard subproblem H2**, your sequence must force **both** of our sample programs to run for at least 10 seconds on our server.

(The hard subproblem would certainly be solvable even if the limits were three times as large. We don't require you to find the worst possible input – any input that's bad enough will do.)

Technical details

For the purpose of this problem, there is no *"the C++"* or *"the Java"*. As the internals of hashsets are implementation-defined, we have to go with specific compiler versions. We did our best to choose the most common ones.

The officially supported versions are **gcc 4.8.1**, **gcc 4.7.2**, **gcc 4.6.4**, **OpenJDK 1.7.0_40**, and **OpenJDK 1.6.0_24**. Solutions that work with any combination of these compilers should be accepted.

Up to our best knowledge, any version of gcc in the 4.6, 4.7, and 4.8 branches should be OK. Any OpenJDK version of Java 6 or 7 should also be OK. However, the only officially supported versions are the ones explicitly given above. For reference, below are sources of two of the officially supported versions.

- gcc 4.8.1
- OpenJDK 7u40
- Should one of the above links fail for some reason, try our local mirror instead.
- You can also find sources of other versions and even browsable repositories on-line if you look hard enough.

Output specification

The file you submit should contain a whitespace-separated sequence of at most 50,000 integers, each between -2^{63} and $2^{63}-1$, inclusive. (Don't worry about the type of whitespace, we will format it properly before feeding it into our Java program.)

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Michal 'Mic' Nánási

Solution

First we shall take a look at how g++ hashes your precious data. The answer will be surprisingly simple. Java does include one more step, but its default hash function is equally broken.

Well, maybe *broken* is a strong word. The default hash functions do often work well in practice. If you are processing your own data, quite often they will perform as expected. That being said, we hope that this task taught you not to trust the default hash functions if your opponents get to choose the data you are hashing. Instead, read up on useful stuff such as *universal hashing*.

What follows is a fairly low-level solution with a lot of technical details. For just the summary, feel free to jump ahead to the last labeled section of this solution.

A general introduction

Hash tables usually work fast. A good hash function distributes the elements evenly across the entire hash table. Whenever a new element arrives, we compute its hash value (i.e., the number of its bucket) and then we compare it only to the elements already present in that bucket.

Ideally, each element should end in a different bucket. In practice, that rarely happens – we have *collisions*, i.e., situations when different elements end in the same bucket.[1] However, we don't need a perfect hash function.[2] All we need is a hash function that is good enough – that is, one that distributes our elements roughly uniformly. Then, it will still be the case that we expect to see a constant number of collisions whenever we insert a new element.

And collisions are precisely what makes a hash table slow. It's easy to see what the worst possible case would be: a hash function that puts all elements into the same bucket. If we insert a sequence of n distinct elements, the hash table will have to compare each pair of elements, which increases the runtime from the nice fast $\Theta(n)$ to the awful $\Theta(n^2)$. That would certainly make you wish you had used a balanced tree with its guaranteed $O(\log n)$ per operation, right?

OK, but certainly the default hash function used in a standard library won't be so dumb... after all, the standard hashsets do actually work well in practice, right? Well, the default hash function in both C++ and Java is only half-bad. It works nicely – as long as the data you are processing behaves nicely. However, as soon as your enemy controls the data, it's pretty easy to push the hashset into the worst possible situation. To achieve that, the enemy will simply select the data in such a way that all elements will hash to the same bucket.

And that is precisely what you had to do in order to score points for this task.

Hashing in C++

Let's take a look at how hashing works in C++. We are using an `unordered_set`. Where can we find it? In the default include directory. On my machine, it's in `/usr/include/c++/4.8.1/unordered_set`.

[1] Collisions are something you have to deal with. Thanks to the *birthday paradox* it is practically impossible to have no collisions at all.

[2] Even though in some very restricted sense such functions do exist – see *perfect hashing*.

Of course, upon opening the `unordered_set` source, we see nothing. Why? Because the true unordered set is hidden in `bits/unordered_set.h`.

Oh, but what's that? Still nothing useful, the unordered set is in fact a `_Hashtable`. And that one can be found in `bits/hashtable.h`. Finally, an implementation of the hashtable!

Here's an interesting piece:

```
// Insert v if no element with its key is already present.
template<typename _Key, typename _Value,
         typename _Alloc, typename _ExtractKey, typename _Equal,
         typename _H1, typename _H2, typename _Hash, typename _RehashPolicy,
         typename _Traits>
  template<typename _Arg>
    std::pair<typename _Hashtable<_Key, _Value, _Alloc,
                                  _ExtractKey, _Equal, _H1,
                                  _H2, _Hash, _RehashPolicy,
                                  _Traits>::iterator, bool>
    _Hashtable<_Key, _Value, _Alloc, _ExtractKey, _Equal,
               _H1, _H2, _Hash, _RehashPolicy, _Traits>::
    _M_insert(_Arg&& __v, std::true_type)
    {
      const key_type& __k = this->_M_extract()(__v);
      __hash_code __code = this->_M_hash_code(__k);
      size_type __bkt = _M_bucket_index(__k, __code);

      __node_type* __n = _M_find_node(__bkt, __k, __code);
      if (__n)
        return std::make_pair(iterator(__n), false);

      __n = _M_allocate_node(std::forward<_Arg>(__v));
      return std::make_pair(_M_insert_unique_node(__bkt, __code, __n), true);
    }
```

This happens to be a part of the machinery that inserts a new element into the hashtable. Once we get over our fear from seeing all those templates (and learn to simply ignore them), the code is pretty readable: take the element, extract its part that is used as the key, hash the key, find the right bucket. If the element is already present in that bucket, just return an iterator pointing to that element, otherwise allocate a new node and insert it into the appropriate place.

Before diving deeper into the source, let's try going closer to the surface instead. Look what we can find in `bits/unordered_set.h`:

```
      /// Returns the number of buckets of the %unordered_set.
      size_type
      bucket_count() const noexcept
      { return _M_h.bucket_count(); }
```

Cool, looks like we can inspect the size of the hash table. That seems useful, let's add it to our sample program:

```
    while (std::cin >> tmp) {
        hashset.insert(tmp);
        std::cout << hashset.size() << "␣";
        std::cout << hashset.bucket_count() << "\n";
    }
```

After running the modified sample program and inserting 50 000 distinct elements, you should see (when using gcc 4.6 or 4.8) the following sequence of hash table sizes: 11, 23, 47, 97, 199, 409, 823, 1741, 3739, 7517, 15 173, 30 727, and 62 233.

The end is the important part. For the last 50 000 − 30 727 inserts (which is almost 20 000 inserts), the hash table size is constant: 62 233. If we can force all elements into the same bucket for this hash table size, we will win: each of the last almost 20 000 inserts will be really really slow.

And we are already half-way there: we know the hash table size. All that remains is the actual hash function: how is the right bucket for an element computed? To answer this question, it's back to the source!

This part of `bits/unordered_set.h` looks promising:

```
      /*
       * @brief  Returns the bucket index of a given element.
```

```
     * @param  __key  A key instance.
     * @return  The key bucket index.
     */
    size_type
    bucket(const key_type& __key) const
    { return _M_h.bucket(__key); }
```

Alright, what is this `_M_h.bucket()`? For that, we have to head over to `bits/hashtable.h` again. First, we shall find the following:

```
    size_type
    bucket(const key_type& __k) const
    { return _M_bucket_index(__k, this->_M_hash_code(__k)); }
```

Oh damn, another layer of wrapper functions. Is that really necessary? But don't worry, we are almost there. One more step in the current file, and then comes another step that will take us to a third file: `bits/hashtable_policy.h`. There we find the following:

```
    std::size_t
    _M_bucket_index(const _Key& __k, __hash_code, std::size_t __n) const
    { return _M_ranged_hash()(__k, __n); }
```

Oh come on! Tell us already! Now all that remains is a sign on the door saying "Beware of the Leopard"! But we will prevail. From here, it's just another few steps down the abyss, and we finally find the well-hidden hash function.

```
  /// Default range hashing function: use division to fold a large number
  /// into the range [0, N).
  struct _Mod_range_hashing
  {
    typedef std::size_t first_argument_type;
    typedef std::size_t second_argument_type;
    typedef std::size_t result_type;

    result_type
    operator()(first_argument_type __num, second_argument_type __den) const
    { return __num % __den; }
  };
```

Wait, so that's it? Oh come on, I could have guessed that two pages ago!

So, after digging around in g++ header files, we come to the stunningly simple conclusion: *Numbers are hashed simply by taking their value modulo the current hash table size. To create an awful lot of collisions, all we need to do is to start inserting numbers that are all, say, multiples of 62 233.*

Some technical details

Here are some technical details you didn't necessarily have to know to solve the task, but it still makes sense to mention them.

First of all, `long long`s are signed, but when hashed their bits are actually interpreted as an unsigned 64-bit integer. For instance, if you start by inserting -1 into an empty `unordered_set<int>` (with 11 elements), it will be hashed to $(2^{64} - 1) \bmod 11 = 4$ and not to $(-1) \bmod 11 = 10$.

Another thing that could trip you up is the mechanism of rehashing. Whenever the hash table size is approximately doubled, all old elements are rehashed into the new table. However, *rehashing runs in worst-case linear time*, even if all old elements get rehashed into the same bucket in the new table. Why? Because the hash table already knows that all values that are being rehashed are pairwise distinct. There is no point in checking that again, so the library doesn't do that.

Finally, the hash table sizes used above are valid for g++ versions 4.6 and 4.8. (Their implementations of the hash table differ in how they rehash, and there is a one-step difference in when they resize the table, but the sequence of hash table sizes, given that you start with an empty table and insert elements one at a time, is the same in both cases.) The implementation used in g++ 4.7 adds 1 before doubling the hash table size, hence a different sequence of hash table sizes is used.

Hashing in Java

In `java/util/HashSet.java` we can find the default settings. Among other things, we can learn that the initial hash table size in Java is 16.

The g++ unordered set resized the table when it was full – i.e., when the number of elements became equal to the hash table size. Java tries to resize sooner: the default "load factor" is 0.75, meaning that the table gets resized whenever it is 3/4 full. (We do not really care about that.)

The rest is simple: a `HashSet` is implemented using a `HashMap`. Next stop: `HashMap.java`. There, we find how the table is inflated when needed:

```
/**
 * Inflates the table.
 */
private void inflateTable(int toSize) {
    // Find a power of 2 >= toSize
    int capacity = roundUpToPowerOf2(toSize);

    threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
    table = new Entry[capacity];
    initHashSeedAsNeeded(capacity);
}
```

Cool, so this time the hash table sizes will be successive powers of two. Even better than in C++.

How do we place an element into the hash table?

```
/**
 * Returns index for hash code h.
 */
static int indexFor(int h, int length) {
    // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";
    return h & (length-1);
}
```

As `length` is a power of 2, the bitwise and is just a fancy way of saying "the bucket is the hash value, modulo table size".

Only one thing remains: what is the hash value of a `Long`? The answer to this question has two steps. First, the `Long` has its own `hashCode`. (This can be found in `java/lang/Long.java`.)

```
public int hashCode() {
    return (int)(value ^ (value >>> 32));
}
```

That is, we take the 32 low bits, xor them with the 32 high bits, and then interpret the result as a signed 32-bit int again.

However, that's not all. Back in `HashMap.java` we can find one more piece of magic. Once we have the `hashCode` of the hashed object, the HashMap shuffles its bits around using the following formula:

```
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();

    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

It turns out that you can ignore the first half, as for the default setting it is equivalent to `h = k.hashCode()`. Thus, all that remains are the last two lines with the magic shifts and xors.

And we are almost done. The last step of this sample solution is to realize that we don't really care what the bitwise magic does. If we compute it for a sufficient number of inputs, we are bound to find 50 000 that *will hash to the same bucket* – in other words, share the last 16 (or maybe 17) bits of the return value. This is easily done by brute force.

Putting it all together

Here is what we learned: In C++, the hash table size is a prime, and a number is hashed to its remainder modulo that prime. In Java, the hash table size is always a power of two, and a number is hashed to its magically shuffled value, modulo the table size.

In order to create collisions in C++, we can simply take any set of multiples of the last table size. Or, for an even better effect, any set of multiples of the last *two* table sizes.

To make sure that the input is also bad for Java, we can simply go through those numbers, hash each of them using the magic function, and only keep those that hash to a 0 when the table size is 2^{16}.

Problem I: Intrepid cave explorer

Maru likes to visit new places, but with her poor sense of direction she always struggles to find her way home. This summer, she's planning to explore an amazing lava cave. Your task is to help her mark the chambers in the cave so that she won't get lost.

Problem specification

The cave consists of n chambers numbered 1 through n. There are $n - 1$ passages, each connecting a pair of chambers in such a way that the entire cave is connected. (Hence, the topology of the cave is a tree.) Chamber 1 contains the entrance to the cave.

Chamber u is an *ancestor* of chamber v if u lies on the path from v to the entrance. (In particular, each chamber is its own ancestor, and chamber 1 is an ancestor of every chamber.) For any chamber $v \neq 1$, the ancestor of v that is directly adjacent to v is denoted p_v and called the parent of v. Chamber numbers are chosen in such a way that for all $v \neq 1$, $p_v < v$.

Maru has two pieces of chalk: a white one and a pink one. She wants to mark each chamber in the cave with some (possibly empty) string of white and pink dots. These strings must satisfy the following requirements:

- If chamber u is an ancestor of chamber v, the string in u must be a prefix of the string in v.
- If chamber u **is not** an ancestor of chamber v and v **is not** an ancestor of u, the string in u **must not** be a prefix of the string in v.

(The empty string is a prefix of any string. Each string is its own prefix. Note that we do not require the strings assigned to chambers to be pairwise distinct.)

Find a valid assignment of strings to chambers that minimizes the total number of chalk dots.

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 a tree. In the first line there is an integer n – the number of chambers. The second line contains $n - 1$ integers $p_2, p_3, \ldots, p_n$ where p_i is the parent of chamber i ($1 \leq p_i < i$).

In the easy subproblem I1, we have $2 \leq n \leq 150$, and no chamber is the parent of more than 20 chambers. In the hard subproblem I2 we have $2 \leq n \leq 21\,000$.

Output specification

For each test case, output one line with one integer – the smallest possible number of dots Maru has to draw in the cave in order to properly mark all chambers.

Example

input

```
1

5
1 1 3 3
```

output

```
6
```

Using `W` *and* `P` *to denote white and pink dots, respectively, one optimal labeling looks as follows:* 1 → "" *(i.e., an empty string)*, 2 → *"*`W`*"*, 3 → *"*`P`*"*, 4 → *"*`PW`*"*, 5 → *"*`PP`*"*.

Task authors

Problemsetter:	Peter 'Bob' Fulla
Task preparation:	Peter 'Bob' Fulla
Quality assurance:	Tomáš 'Tomi' Belan

Solution

The solution for I will be added later.

Problem J: Judging programs

Each subproblem of this problem is to be solved separately. They both share the same principle: you are given a formal specification of a task and a set of programs. Each of those programs contains a function named `solve` that attempts to solve the given task. All programs are written in an easily-readable subset of Python, and shouldn't contain any language-specific tricks or gotchas.

Your goal is to determine which of those programs correctly solve the given task. We consider a program correct if, for every valid input, the program terminates after a finite number of steps (even if that number of steps is huge) and the returned answer is correct.

Easy subproblem J1

Given are two integers n and k ($1 \leq k \leq n \leq 100$). The task in this subproblem is to compute the value of the binomial coefficient $\binom{n}{k}$. In other words, compute the number of ways in which we can choose a k-element subset of an n-element set. For example, for $n = 10$ and $k = 2$ we should compute the value 45. (Note that Python can handle arbitrarily large integers.)

The programs are stored in files named `j1_01.py` through `j1_11.py`. Each program contains a function `solve` that expects 2 parameters: the integers n and k.

The output file you submit as your solution to the easy subproblem J1 must contain exactly 11 whitespace-separated words. The i-th word of your output describes the i-th program `j1_i.py`. The word must be "`good`" if the program correctly solves all valid instances, or "`bad`" otherwise.

Hard subproblem J2

Given is a simple connected undirected weighted graph. The task of this subproblem is to find the length of the shortest path between two given vertices of the graph.

The programs are stored in files named `j2_01.py` through `j2_15.py`. Each program contains a function `solve` that expects 5 parameters:

- `n`: the number of vertices in the graph ($2 \leq n \leq 100$). Vertices are numbered 0 through $n - 1$.
- `m`: the number of edges in the graph ($n - 1 \leq m \leq n(n - 1)/2$).
- `edge_list`: the list of all m edges of the graph. Each edge is a triple (p, q, w) where p and q are the vertices that are connected by the edge ($p \neq q$) and w is a positive integer which denotes the edge length ($1 \leq w \leq 1\,000$). Note that no two edges share the same pair $\{p, q\}$.
- `start` and `end`: the two (distinct) endpoints of the path we are looking for.

For your convenience, we also provide the files `j2_sample.{in,out}` and `j2_read_input.py`. The `in` file contains a sample instance, the `out` contains its correct solution. The third file contains the function `read_input` that loads an instance from standard input and parses it into arguments expected by `solve`.

The output file you submit as your solution to the hard subproblem J2 must contain exactly 15 whitespace-separated words. The i-th word describes the i-th program `j2_i.py`. The word must be one of "`good`", "`ones`", and "`bad`". Here, "`ones`" means that the program is *not* correct in general, but it does correctly solve all instances with unit-length edges (i.e., ones where for each edge we have $w = 1$). Answer "`bad`" if neither "`good`" nor "`ones`" applies.

Submission count limits and grader responses

If you make a submission that is syntactically correct but does not get all answers right, our grader will tell you how many programs were classified correctly.

Note that you can only make **at most 7 submissions** for the easy subproblem J1, and at most 10 submissions for the hard subproblem J2. Be careful.

Task authors

Problemsetter:	FIXME
Task preparation:	Monika Steinová
Quality assurance:	FIXME

Solution

Easy subproblem

The task in this subproblem is to compute the binomial coefficient $\binom{n}{k}$.

Using well-known formulas

The programs `j1_01.py`, `j1_03.py`, `j1_04.py`, `j1_05.py`, `j1_07.py`, `j1_08.py` and `j1_11.py` compute the binomial coefficient via a (sometimes modified) well-known formula.

In `j1_01.py` the recursive formula is used and program's output is always correct. The binomial coefficient $\binom{n}{k}$ is expressed from this formula for $\binom{n+1}{k}$ in `j1_07.py`. This program is incorrect as it never terminates. This is due to increasing the first argument of the first call of `solve` in line 3 and thus missing the termination condition.

A modified, but still correct, version of Vandermonde identity is implemented in `j1_03.py`. It is based on the following observation: In how many ways can we select k elements out of n? Let's divide the elements into $n - step$ in one group and $step$ in another group. Then, for some i, we have to select i elements from the first and $k - i$ from the second group.

Another famous identity is the Hockey-Stick identity which is invoked in `j1_04.py` and `j1_05.py`. Its standard implementation in `01_04.py` is correct. The version in `j1_05.py` is incorrect (e.g., $n = 21$ and $k = 1$) – the first argument of `solve` function ranges between $n - 20$ and $n - 1$ (inclusive), but in the correct identity the ranges are between $k - 1$ and $n - 1$ (inclusive).

The computation of the binomial coefficient via expansion to factorials is implemented in `j1_08.py` and the program is correct. Using the factorial expansion one can express the binomial coefficient as $\binom{n}{k} = \binom{n}{k+1} \cdot \frac{k+1}{n-k}$ which is implemented in `j1_11.py` and is also correct.

Computing counts of seemingly unrelated objects

Perform n steps in a grid starting in point $(0, 0)$ walking in the 4 possible directions. How many of such different walks will end up in the point $(k, n - k)$? This is a question which solves program `j1_02.py`. Since to get to point $(k, n - k)$ from $(0, 0)$ one needs exactly n steps, in all the walks terminating in $(k, n - k)$ one has to always move towards point $(k, n - k)$ and never walk back. This is a standard combinatorial problem in which the number of such paths sums to $\binom{n}{k}$. Our program computes exactly this number and thus it is correct.

The program `j1_06.py` stores in buckets `A[k]` the count of numbers from 0 up to $16^n - 1$ (inclusive) that have k odd digits in their hexadecimal representation. For each possible sequence of n bits containing exactly k ones there are exactly 8^n such numbers counted in `A[k]` – in each hexadecimal digit the first 3 bits can be picked arbitrarily (that is 8 combinations) and the last one has to be set to 1 to obtain an odd digit. Thus `A[k] // 2**{3*n}` is equal to `A[k] // 8**n` and the output of the `solve` function is identical with the count of the numbers from 0 to $2^n - 1$ (inclusive) with k ones and $n - k$ zeros in their binary representation. This is nothing else than counting the number of ways how from n element set one can pick k elements and that is one of possible definitions of $\binom{n}{k}$. Thus the program `j1_06.py` computes a correct solution.

A pseudo-random coin toss is implemented in `j1_09.py`. The pseudo-random numbers are generated and stored in the variable `state`. From this variable n coin tosses are read and the number of seen heads is determined (lines 6 to 9). If exactly k heads are seen, the global counter `good` is incremented. The tossing and reading of the coins corresponds to selecting a set of k elements from a set of n elements. Hence if such tossing is repeated 2^n-times the variable `good` is in average $\binom{n}{k}$. Since the program iterates $8^n = 2^n \cdot 4^n$ times (line 4), the returned result has to be normalized by dividing with 4^n. The program is incorrect, if we choose inputs that are large enough, many of them will actually fail due to insufficient precision.

Another classical problem in combinatorics is to count the number of correctly parenthesized expressions. In `j1_10.py` such expressions of length $2n$ that have a maximum nesting depth of k are counted. The result has nothing in common with binomial coefficient and thus program `j1_10.py` is incorrect (e. g., for $n = 4$ and $k = 3$).

Hard subproblem

In this subproblem given is a simple connected undirected weighted graph and the goal is to find a shortest path between two given vertices `start` and `end`.

Floyd-Warshall algorithm and its variations

A standard algorithm for computing the shortest path between all pairs of vertices of a graph is Floyd-Warshall algorithm. The J2 programs `j2_01.py`, `j2_06.py`, `j2_09.py` and `j2_12.py` implement similar algorithms. The search of the shortest path backwards with this algorithm is implemented in `j2_01.py` and `j2_06.py`. Both programs are correct as the graph is undirected and thus `edge[i][j]` and `edge[j][i]` are symmetric. The file `j2_09.py` contains a classical implementation of Floyd-Warshall algorithm which is again correct. The only incorrect implementation is in `j2_12.py` – the order of indices in the loops is swapped.

Dijkstra's algorithm and its variations

To compute a point-to-point shortest path in a weighted graph without negative cycles often Dijkstra's algorithm is used. Various implementations of this algorithm are exploited in `j2_02.py`, `j2_05.py`, `j2_13.py` and `j2_15.py`.

The traditional implementation of the algorithm is in programs `j2_02.py` and `j2_13.py` which differ in termination condition. In `j2_02.py` the program terminates once the priority queue is empty. In `j2_13.py` the function `solve` terminates once the endpoint is picked from the queue. There are two well-known invariants valid in such implementation:

- The vertices `u` are popped from the priority queue by increasing distance from the starting vertex.
- Once the vertex `u` is popped from the queue its distance is final and will not decrease later.

Due to the second invariant both `j2_02.py` and `j2_13.py` terminate with a correct result.

Next, there is Prim's algorithm which is used to build a minimum spanning tree and which is the basis for `j2_05.py` and `j2_15.py`.

Maybe a bit surprisingly, the version in `j2_15.py` correctly computes the shortest path between `start` and `end`. Note that `j2_02.py` and `j2_15.py` differ only in line 23 in the priority of items inserted into the priority queue. The incorrect priorities make `j2_15.py` slower than `j2_02.py`, but still correct. The reason is that for any shortest path P from `start` to a vertex x it is possible to prove that each vertex in this path will eventually have computed the correct minimum distance from `start`. When a vertex from P is processed and its minimum distance from `start` is correctly set, the loop in lines 12 to 23 inserts to the queue all the incident edges. This is also true for the edge leading to the next vertex in P which is processed in the loop and the minimum distance to the following vertex in P is computed correctly. Eventually, again the following vertex in P is popped from the priority queue. The loop inserts another vertex of P to the queue and updates the distance to the minimum one and so on.

The program in `j2_05.py` is the version of `j2_15.py` which the termination condition of `j2_13.py`. Clearly this program is incorrect as it terminates once the `end` vertex is popped from the priority queue which, as discussed above, might be premature.

Bellman-Ford algorithm

Only program `j2_08.py` implements a version of Bellman-Ford algorithm which can be used to compute the shortest path from a single vertex to all other vertices. Our implementation is incorrect as the outer loop uses $n-2$ instead of the minimum necessary $n-1$ iterations. For graphs where the shortest path contains $n-1$ edges and the edges are presented to the algorithm in reverse order, the algorithm will fail.

Breath-first search algorithm

We did not omit the classical BFS algorithm which is implemented in `j2_10.py` and `j2_14.py`. The programs differ in the point where the processed vertex is marked. In `j2_10.py` the processed vertex is marked once it is discovered. This is a standard and correct way of implementing BFS and thus the program is correct but only works for graphs with edge-weights one (i.e., $w = 1$).

The second implementation in `j2_14.py` is completely incorrect – the problem occurs if the distance of a vertex is multiple times changed (line 24) before it is popped from the queue and marked (which then disallow to update the distance further to something better).

Other techniques

A standard backtrack without recursion is implemented in `j2_03.py` and is correct. The implementation uses two stacks – one for storing the visited vertices in the current path (`vertex_stack`) and another one for storing the index of the edge which has to be searched through the next time the backtrack returns to the corresponding vertex.

The program in `j2_04.py` is also correct and it is an implementation of Savitch's theorem. The shortest path between `start` and `end` can have at most `magic` ($> n$) steps. Thus the algorithm tries each vertex to be the middle vertex of such path (line 10) and recursively finds two paths of half that length – one between starting and middle vertex and the other one between middle and ending vertex. From all such paths the algorithm picks the one with is the shortest. The recursion terminates if the starting and ending vertex are identical or connected by an edge.

The file `j2_11.py` implements a simple heuristic: a search from one of the endpoints to the other one by visiting vertices via the cheapest edge leading to not yet visited vertex. Such a heuristic is very wrong and a counterexample can be found easily.

Finally, `j2_07.py` is pretending to be a program that generates a lot of pseudorandom paths. However, this program was trying to trick you. The "pseudorandom generator" is, in fact, a binary counter, and the way it is used makes sure that each possible path is tried at least once.

Problem K: Knowledge test

Programming competitions are great! Even if you don't know any encyclopedic facts, you can still successfully solve many problems if you are smart. But that won't be enough in this problem.

Problem specification

You are given a crossword puzzle. Your task is to solve it.

A crossword is a rectangular grid consisting of black and white squares, and a set of clues. Each clue is an English sentence that describes a single word. For each clue, we are also given the coordinates of a white square in the grid and a direction – either horizontally ("across", meaning from left to right) or vertically ("down", meaning from top to bottom). The goal is to write a letter into each white square in such a way that for each clue, we can read its solution in the grid by starting at the given square and reading in the given direction. Black squares are only used to separate the words in the grid.

Note that the clues precisely correspond to all maximal consecutive groups of two or more white squares in rows and columns of the grid. (There are no clues with 1-letter answers.)

Input specification

The input file contains exactly one crossword puzzle to solve.

The first line contains two integers r and c: the number of rows and columns of the grid. Rows are numbered 0 to $r-1$ starting at the top, and columns are numbered 0 to $c-1$ starting on the left.

Each of the next r lines contains c characters. Hashes ("`#`") represent white squares, dots ("`.`") represent black squares.

The next line contains one integer a, specifying the number of clues with answers written across. Each of the next a lines contains two numbers r_i and c_i, and a clue. Here, (r_i, c_i) are the coordinates of the leftmost letter of the clue's solution.

The next line contains one integer d, specifying the number of clues with answers written down. Each of the next d lines contains one clue in the same format as above, with (r_i, c_i) being the topmost letter.

The crossword has a unique correct solution.

Output specification

Take the crossword from the input and replace all `#` characters with lowercase English letters (`a-z`). The output file must contain exactly r lines, each exactly c characters long.

Example

input

```
3 4
..#.
####
..#.
1
1 0 Your favorite competition
1
0 2 Traveling salesman problem acronym
```

output

```
..t.
ipsc
..p.
```

Task authors

Problemsetter:	Michal 'Mimino' Danilák
Task preparation:	Michal 'Mimino' Danilák
Quality assurance:	Michal 'Mic' Nánási

Solution

You can solve the crossword puzzles in three steps. First two steps are fairly implementational. The third step is then pretty straightforward.

Firstly, by looking at the input files you have to identify, download and parse out all the information required to solve the clues. For example you can use the following websites as the source of information:

- Pokémons: http://en.wikipedia.org/wiki/List_of_Pok%C3%A9mon
- US presidents: http://en.wikipedia.org/wiki/List_of_Presidents_of_the_United_States
- Chemical elements: http://en.wikipedia.org/wiki/List_of_elements
- Web colors: http://www.w3schools.com/html/html_colornames.asp

There are many different ways to parse the above webpages into some useful format. Depending on your skills you can either use regular expressions, HTML parsing libraries or some UNIX-fu.

Secondly, you have to parse the given clues and for each clue collect the list of possible answers (yes, some clues can have multiple correct answers). Regular expressions should be your weapon of choice for this task.

In hard subproblem, some numbers are written in the form of expressions which you have to evaluate to get the number from the clue. You can either write your own parser or if you use Python, just call `eval()` function on the expression.

Finally, after we collected for each clue the list of possible answers, it is time to solve the crossword puzzle. Actually this step is the easiest one and following simple algorithm is enough to solve the whole puzzle:

```
while crossword not solved:
    choose a clue with only one answer that can be filled
    fill the answer
```

Problem L: Let there be sound

In each subproblem, we have given you an MP3 file. Listen to it. Analyze what you hear. Follow the trail we left for you. Sooner or later, you should somehow extract the answer: a positive integer.

Because of their size, the MP3 files aren't in the `ipsc2014` archives. Download them from the online problem statement.

(In case you are listening to the easy subproblem right now: no, that's not the integer you are looking for. That would be too easy, wouldn't it?)

Output specification

Submit a text file containing a single positive integer (e.g., "`47`").

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek
Quality assurance:	Martin 'Imp' Šrámek, Michal 'Mic' Nánási

Solution

In the easy subproblem the MP3 file seems to contain a lot of zeros and ones. In the hard subproblem we have mushrooms, snakes, and… are those badgers?

Easy subproblem

In the easy subproblem we hear a lot of zeros and ones (courtesy of `espeak`). And then, close to the end, there is one exception: suddenly we hear the number 947 012 399. (Probably the easiest way to discover it: open the MP3 file in some editor and look at the waveform.)

Still, the statement warns us that this is not the answer yet. What shall we do? Well, the zeros and ones quite obviously come in groups of eight, so maybe they are bits grouped into bytes. We "just" have to convert them into a more conventional representation.

How to do that painlessly? We will enlist the help of some standard tools. One such tool is mp3splt. We can use it to split our MP3 file into many smaller ones. The best way is to use mp3splt's built-in functionality: split whenever there is silence. Our reward: about a thousand smallish mp3 files.

One of those files will be slightly larger than the others – that is the special one with the nine-digit number. Each of the other files will contain either a single "zero" or a single "one". How can we tell them apart?

Simply. We can do it without any speech recognition, and even without comparing samples. Instead, we can simply look at the file sizes. The word "zero" is longer than the word "one", hence the files that contain "zero" are larger. On our machine, "zero"s turned out to have about 2.2 kB while "one"s only had about 1.8 kB each.

Having those zeros and ones, we can easily convert them into characters and read the message:

Hello! As your answer, submit the largest prime factor of the following number: 947 012 399*. We wish you good luck!*

We have $947\,012\,399 = 9\,103 \times 104\,033$, hence the answer we seek is 104 033.

Hard subproblem

In the hard subproblem we had two separate parts. First, you had to parse a sound file into badgers, mushrooms, and snakes. You also had to guess the encoding used, but that should have been reasonably easy: there are at most two consecutive snakes, at most four consecutive badgers, at most three consecutive mushrooms. Moreover, sequences of badgers+mushrooms never exceed four elements. That should have been a sufficient indicator that we are using Morse code: badgers are dots, mushrooms are dashes, and snakes serve as separators between characters and words.

There are several ways how to parse the sound file, for instance FFT can be used to match the samples to each other. However, there are also simpler ways. For example, cross-correlation should be sufficient. To use cross-correlation, convert the MP3 into a wave, and then try computing the correlation between a prefix of the input and other parts of the wave. Once you find a part that matches, you have identified the first sample (or two), and you can also autodetect its approximate length. Save it into a separate file and cut it off. Repeat the same process until you have everything split into tiny files. (Of course, in the next rounds it is best to start by comparing the prefix of your current wave to the samples you already identified.)

Once you got this far (or if you were really patient and translated the Morse code manually), you should have seen the following message: "`this is indeed morse code you got it now go to youtube dot com and watch the video with the following id l l capital a n capital s 7 p t capital l e capital m good luck and have fun`".

The URL described in the solution did really work and if you entered it into your browser, you saw this video.

The video clip has two separate tracks: audio and video. The audio track uses exactly the same encoding as before and hides the following message: "`and just when you thought it was over here is another level oh wait make that two levels there are also the moving pictures deal with those then add forty-seven to the result`".

The video is completely independent from the audio track (although this may be hard to notice at the first glance :)) and it uses exactly the same encoding.

Parsing the video is even easier than parsing the audio track. All you need to know: badgers are green, mushrooms are red, snakes are yellow. We don't really care whether there is one snake or two. Each badger in the video takes the same amount of time. There are only two different mushrooms and they always alternate. So it's enough to sample static frames from the video and classify them according to their color. This gave you the final message: "`wow you must be very good take the youtube id of this video convert it to all uppercase and then interpret it as a number in base thirty-four afterwards apply what you learned from the sound of this video and voila you are done`".

The video id in all uppercase is LLANS7PTLEM. If we interpret it as a number in base 34, and then add 47 (base 10) to the result, we get the final answer: 44 646 099 266 597 965.

Problem M: Maximum enjoyment

As a grad student researching computer networking, Peter knows a lot about new networking trends. For example, take Multipath TCP ("MPTCP") – an extenstion of the standard TCP/IP protocol which can send data over multiple paths. While the main purpose of MPTCP is to provide a seamless connection between different networks (such as Ethernet, Wi-Fi and 3G), it can also be used to improve connection bandwidth.

And today, such an improvement would be really handy. Peter is far from the IPSC headquarters at Comenius University, but he would still like to see what the other organizers are doing (desperately fixing last minute issues, eating popcorn and laughing at your submissions). To watch all of this in detail, Peter wants to set up a video call of the highest possible quality. Therefore, he plans to use use all available bandwidth between his home and Comenius University by splitting the video stream over several MPTCP subconnections, each having its own path.

Problem specification

You are given an undirected graph of network routers and links. Your task is to calculate the maximum amount of traffic Peter can stream from the source router (located at Comenius University) to the sink router (located at Peter's home). To send the traffic, Peter can configure MPTCP by selecting a set of (not necessarily disjoint) paths through the network and specifying a desired bandwidth on each of them as an arbitrary-precision floating point value. The total bandwidth of the video stream will be the sum of these per-path bandwidths. But each link has a maximum throughput, or *capacity*, that cannot be exceeded.

So far, this looks like a typical max-flow problem. Unfortunately, Peter wants to stream live video, and video is very sensitive to latency. This means Peter can only send data through paths that have no more than L links.

Given this constraint, find the maximum bandwidth Peter can achieve.

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 a single line containing four integers: the size of the network N ($N \geq 2$), the index s of the source router ($0 \leq s < N$), the index t of the sink router ($0 \leq t < N, s \neq t$), and the maximum path length L ($L \geq 1$). (Routers are indexed from 0 to $N - 1$.)

The next N lines describe the links between routers and their capacities. Indexing from zero, the i-th line contains n integers $c_{i,j}$ ($0 \leq c_{i,j} \leq 10000$) specifying the (directional) link capacity (in megabits per second) between routers i and j. The adjacency matrix is symmetric ($c_{i,j} = c_{j,i}$) and the diagonal only contains zeros ($c_{i,i} = 0$).

- In the easy subproblem M1, $t \leq 100$, $L \leq 3$, and $N \leq 100$.
- In the hard subproblem M2, $t \leq 30$, $L \leq 6$, and $N \leq 100$.

Output specification

Internet service providers like to give speeds in megabits per second because it makes the numbers look bigger, but end users mostly care about bytes, not bits. So for each test case, output a single line with the maximum streaming bandwidth given as a floating point number **in megabytes per second**. Any answer within a relative error of 10^{-9} will be accepted.

Example

input

```
3

3 0 1 1
0 7 5
7 0 3
5 3 0

3 0 1 2
0 7 5
7 0 3
5 3 0

5 0 4 3
0 2 2 0 0
2 0 9 2 0
2 9 0 2 0
0 2 2 0 3
0 0 0 3 0
```

output

```
0.875
1.25
0.375
```

In the first test case, Peter can only use a direct path from 0 to 1 with capacity 7 Mbit/s. This gives 0.875 Mbytes/s. In the second test case, Peter can setup MPTCP to follow both the direct path and the path $0 \to 2 \to 1$ *with an additional capacity of 3Mbit/s. In the third test case, Peter needs to split the traffic across two paths, but they will share a bottleneck between routers 3 and 4.*

Task authors

Problemsetter:	Peter 'PPershing' Perešíni
Task preparation:	Peter 'PPershing' Perešíni
Quality assurance:	Vlado 'usamec' Boža

Solution

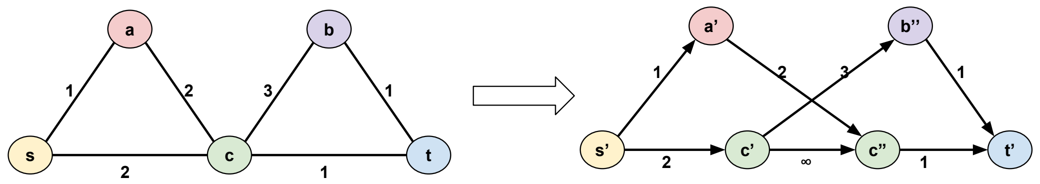
We will start with the easy subproblem. For $L = 1$ and $L = 2$ the situation is trivial: In the first case we can use only direct path and in the second case we can simply compute $result = c_{s,t} + \sum_v \min(c_{s,v}, c_{v,t})$ as all available paths are edge-disjoint (Here s and t denote sink and source node respectively).

The situation is more tricky for $L = 3$ — the remaining value of L present in the easy input — as paths of length 2 and paths of length 3 may share the same edge from sink or towards source. To deal with the issue, let us first consolidate path lengths by extending each path of length 2 to a path of length 3 by introducing a dummy self-loop. In particular, we will introduce a self-loop in the middle vertex of the path (e.g., the only vertex of a path $\neq s, t$). This will result in two types of paths:

1. paths of the form $s \to u \to v \to t$, and
2. paths of the form $s \to w \to w \to t$.

We may notice that in both cases, the first vertex on the paths is s, the second vertex is a neighbour of s, third vertex is a neighbour of t and fourth is t.

Based on our observation, we may transform the original graph into new **directed** graph G' as follows: First, create new source and sink nodes s' and t' (and add a possible direct edge between them if it existed in the original graph). Afterwards, for each neighbour u of s in G, we add a new vertex u' to the graph G' and add edge $s' -> u'$ with capacity of the edge $s \to u$. We repeat a similar process with sink – we add each neighbour v of t in G as a vertex $v'' \in G'$ and add edge $v'' -> t'$ with capacity of $v -> t$. Now we need to connect neighbours of s to neighbours of t, i.e., add edges $u' \to v''$ iff there was a link between u and v in the original graph (again with the corresponding capacity). Finally, we need to solve self-loops — if a vertex w is both neighbour of s and t, we need to add additional edge $w' \to w''$ with an unlimited capacity specifying the self-loop. If you are confused about the construction, here is an example:



Now, if we look at G', any path from s' to t' actually corresponds to path of length 2 or 3 in the original graph and the paths share the same bottlenecks in the both graph. Thus, by solving an unrestricted maxflow problem on G', we will directly get the solution of $L = 3$ restriction in the original graph G.

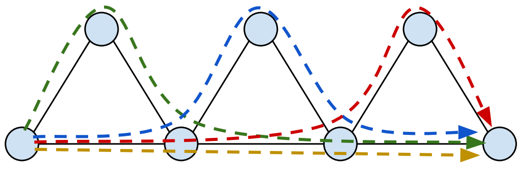
Hard subproblem

It would be tempting to extend the solution from the easy subproblem to $L > 3$ as well. Unfortunately, the trick we used does not work here — if we setup G' for $L = 4$ such that there would be 5 layers:

1. source s
2. neighbours of s
3. neighbours of both {neighbours of s} and {neighbours of t} at the same time
4. neighbours of t
5. sink t

we would get into trouble in graph G' as an arbitrary edge $u \to v$ in G might be now represented by multiple edges in G'. Therefore, to honor the capacity constraints, these edges would need to have shared bottleneck (e.g. sum of their flows cannot exceed capacity $c_{u,v}$). Unfortunately, such constraint is impossible to emulate in standard maxflow setting.

Moreover, you probably noticied something weird about output formulation of the problem statement — why would we want to convert from Mbit/s to Mbyte/s? The answer is perhaps obvious if you know we are tricksters — we simply wanted to hide the fact that the solution might not be an integer. As an example, take the previous graph but extend it with one more triangle and require paths of length ≤ 4. There are 4 available paths in the graph: **g**reen, **b**lue, **r**ed and **y**ellow.



The restrictions for the bandwidth along these paths are as follows:

- $r + \quad + b + y \leq 1$
- $r + g + \quad + y \leq 1$
- $\quad + g + b + y \leq 1$

Putting it together, $2(r + g + b + y) \leq 3 - y$ which immediatelly implies that $r + g + b + y \leq 1.5$ as $y \geq 0$ (we can send only positive amount of bandwidth over a path). Therefore, one of the optimal solution solution to the problem is $y = 0, r = g = b = 0.5$ and results in non-integer total bandwidth of 1.5 units/s. Moreover, the solution exhibits even more counter-intuitive property — apparently in this case it is beneficial to take longer paths (r, g, b) instead of the short one (y).

The previous example is bad news for us – any max-flow like algorithm (or heuristic) would surely produce only integral solutions and thus cannot work properly. However, we can still learn something from it. Namely, we can repeat the same thing with an arbitrary graph — we will enumerate all paths, describe each path as a positive unknown variable and pose a set of constraints, one for each edge of the graph. This should sound familiar as it smells quite strongly with linear programming.

Unfortunately, this observation was not enough to solve the hard subproblem. The main problem lies in testcases with $L = 6$ which can have (read: we made sure they have) several millions of paths. This produces linear programs which are near unsolvable during the competition time (our sequential solution took 8 hours to solve all test cases and consumed excessive amounts of memory). Thus, we need an alternative LP formulation of this problem.

We start by revising the standard LP formulation of the maxflow problem. In this formulation, we have a variable $x_{u,v}$ for each edge $u \to v$ specifying how much flow is flowing through it. For the reverse direction, we have $x_{v,u} = -x_{u,v}$. The **capacity constraints** are $-c_{u,v} \leq x_{u,v} \leq c_{u,v}$. Additionally to capacity constraints, we have **flow conservation constraints**: For each vertex $u \neq s, u \neq t$, we must ensure that $0 = \sum_v x_{u,v}$. The objective is simply to maximize flow ourgoing from the source, e.g. $maximize : \sum_u x_{s,u}$.

The standard maxflow formulation, however, does not care about paths at all and thus cares even less about their constraint on length being $\leq L$. To emulate the path length constraint, we can reuse the idea we already had after easy subproblem — let us split each vertex into $L + 1$ “layers” describing how far are we from the sink on the path. Then, taking an edge would meen going one layer up. For example, if we had path $s \to u \to v \to t$, we would take vertices s_0, u_1, v_2 and t_3. Of course, this would mean that the capacity constrints are shared between several copies of the same edge on different layers but now with LP we can easily deal with these linear constraints.

This gives as a recipe on how to solve the constrained maxflow problem using linear programming:

1. Split each vertex into $L + 1$ copies numbered from 0 to L.
2. For each edge $u \to v \in G$ and for each level $0 \leq i < L$ add a directed edge from vertex u_i to vertex v_{i+1} and denote the amount of flow on this edge as $x_{u,v,i}$ where $x_{u,v,i} \geq 0$.
3. Capacity constraints:
   $$\forall u \to v \in G : \sum_{i, 0 \leq i < L} x_{u,v,i} \leq c_{u,v}$$
4. Flow conservation constraints for each splitted vertex:
   $$\forall v \in G, 1 \leq i < L : \sum_{u, u \to v \in G} x_{u,v,i-1} = \sum_{w, v \to w \in G} x_{v,w,i}$$
5. The objective is to generate as much flow at sink as possible:
   $$Maximize : \sum_{u, s \to u \in G} x_{s,u,0}$$

There are, however, two ways in which LP can trick us right now into giving us a solution we do not want: It can start flow at level 0 anywhere and also end it anywhere. Moreover, because there is no flow conservation law for sink and source, the solver can start a flow at the source and then loop it back to the source. While, in principle, it is not prohibited to loop the flows in the problem statement, it directly interferes with the fact that our objective measures outgoing flow from source at level 0 instead of measuring total $s \to t$ flow. To fix these issues, we need to add following constraints as well

6. Flows cannot start in any vertex of G except for the source:
   $\forall u, u \in G, u \neq s : \forall v, u \to v \in G : x_{u,v,0} = 0$
7. Flows cannot end in any vertex other than sink:
   $\forall u, u \in G : \forall v, u \to v \in G, v \neq t : x_{u,v,L-1} = 0$
8. Disallow sink self-loops. It not necessary to self-loop in an optimal solution and thus we will require that no flow flows to the sink:
   $\forall u, u \to s \in G : \forall i, 0 \leq i < L : x_{u,s,i} = 0$

Solving the aforementioned linear program after formulating it is easy — you can use any off-the-shelf available solver (e.g. lpsolve, SCIP, CPLEX, etc.). Here we just note that we advise against using lpsolve command line tool and use API instead as the commandline tool rounds the results to roughly 7 digits which less less than required precision. Or, you can simply write your own simplex algorithm — the testcases here are small enough to be solvable even by hand-written implementations. The only needed optimization in our Python+Numpy solution is a presolving step which removes all variables equal to zero (e.g. skip edges with zero capacity).