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.

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.

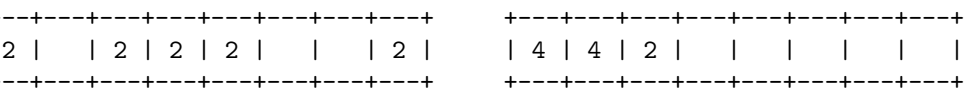
The game ends when there is no move that would change the game state.

Moves and merges

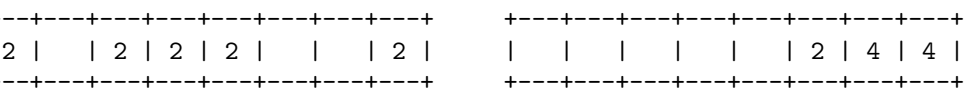
When the player chooses a direction for tile movement, all tiles move in that direction one after another, starting with the tile that is closest to the target boundary. Whenever a tile collides into another tile that also has the same integer, the two tiles merge into one tile that has the sum of their 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.



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.



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
```


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 $a_0, a_1, a_2, \ldots, a_{m-1}$. The copier chooses some i, j, k such that $0 \leq i < j \leq k \leq m$. Then the copier inserts a copy of $a_i, \ldots, a_{j-1}$ immediately after a_{k-1}. Note that $j \leq k$: the copy is always inserted to the right of the original. Here is how the sequence looks like after the insertion:

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

Problem specification

In the morning we had some **permutation** of $1 \ldots \ell$ 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 $t \leq 60$ 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 ($3 \leq n \leq 100\,000$) – 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 $\{1, 2, \ldots, \ell\}$ using a finite sequence of copier operations.

In the **easy subproblem C1** you may also assume that $n \leq 10\,000$, 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)`.

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.

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 $128 \times 128 \times 3$ 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
```

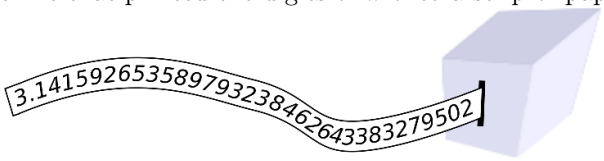
This is a valid program that produces an answer for each image. But the answers aren't correct, so you'd receive this response:

```
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.

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.)

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`.

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.)

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	output
1 5 1 1 3 3	6

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

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.

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.
```


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 large integer that ends in 99 is not what 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`" – quotes for clarity).

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.*

