# **Exercise 1: ⬤**

You are given words. Some words may repeat. For each word, output its number of occurrences. The output order should correspond with the input order of appearance of the word. See the sample input/output for clarification.

**Note:** Each input line ends with a **"\n"** character.

**Constraints:**

1 ≤ n ≤ 105

The sum of the lengths of all the words do not exceed 106 All the words are composed of lowercase English letters only.

## **Input Format**

The first line contains the integer, **n**.

The next **n** lines each contain a word.

## **Output Format**

Output 2 lines.

On the first line, output the number of distinct words from the input.

On the second line, output the number of occurrences for each distinct word according to their appearance in the input.

**Sample Input:**

4 bcdef abcdefg bcde bcdef

## **Sample Output**

3

2 1 1

## **Explanation**

There are 3 distinct words. Here, **"bcdef"** appears twice in the input at the first and last positions. The other words appear once each. The order of the first appearances are **"bcdef"**, **"abcdefg"** and **"bcde"** which corresponds to the output.

Python code:

n = int(input())

# dictionary to store word counts

word\_counts = {}

for i in range(n):

word = input().strip()

if word in word\_counts:

word\_counts[word] += 1

else:

word\_counts[word] = 1

print(len(word\_counts))

for word in word\_counts:

print(word\_counts[word], end=' ')

print()

# 

According to this code, the integer “n” from input. After shat word\_counts stands as created dictionary in order to store the counts of each word.

We then use loop over the next n lines of input, processing each word. For each word, we strip any leading or trailing whitespace using the strip() method, and compare word\_counts to understand whether it already exists or not. If it do exists we increase its count by 1. If it doesn't exist , we add it and then have its count as 1 in word\_counts.

After processing all the input words, we output the number of distinct words in word\_counts using the len() function. Then we again use loop over the keys of word\_counts (which are the distinct words) and output their counts in the order they first appeared in the input.

After that with the help of end parameter of of the print() function to output the counts on a single line separated by spaces, and then print a newline character (\n) at the end to move to the next line.

# **Exercise 2: ⬤**

**Lexicographical order** is often known as alphabetical order when dealing with strings. A string is greater than another string if it comes later in a lexicographically sorted list.

Given a word, create a new word by swapping some or all of its characters. This new word must meet two criteria:

● It must be greater than the original word

● It must be the smallest word that meets the first condition

## **Example w = abcd**

The next largest word is **abdc**.

Create the function bigger\_Is\_greater and return the new string meeting the criteria. If it is not possible, return **no answer**.

## **Function Description**

Function has the following parameter(s):

● string w: a word

**Returns**

- string: the smallest lexicographically higher string possible or **no answer**

## **Input Format**

The first line of input contains **T**, the number of test cases. Each of the next **T** lines contains **w**.

**Constraints**

● 1 ≤ **T** ≤ 105

● 1 ≤ length of **w** ≤ 100

● w will contain only letters in the range ascii[a...z]

**Sample Input:**

5 ab bb hefg dhck dkhc

## **Sample Output** ba no answer

hegf dhkc hcdk

**Explanation** Test case 1: ba is the only string which can be made by rearranging ab. It is greater.

Test case 2:

It is not possible to rearrange bb and get a greater string.

Test case 3: hegf is the next string greater than hefg.

Test case 4: dhkc is the next string greater than dhck. Test case 5: hcdk is the next string greater than dkhc.

**Sample Input:** 6 lmno dcba dcbb abdc abcd fedcbabcd

**Sample Output** lmon no answer no answer acbd abdc Fedcbabdc

# 

Python code:

def biggerIsGreater(w):

n = len(w)

pivot = n - 2

while pivot >= 0 and w[pivot] >= w[pivot + 1]:

pivot -= 1

if pivot < 0:

return "no answer"

successor = n - 1

while successor > pivot and w[successor] <= w[pivot]:

successor -= 1

w = list(w)

w[pivot], w[successor] = w[successor], w[pivot]

w[pivot + 1:] = sorted(w[pivot + 1:])

return ''.join(w)

T = int(input())

words = []

for I in range (T):

  word = input().strip()

  words.append(word)

for word in words:

  print(biggerIsGreater(word))

in this code we look for the pivot point starting from the end of the string and define the first pair of it, character on the left is smaller than on the right. If we can't find such a pair, then it means "no answer".But if we have found the pivot point, we find the character to the right of the pivot that is greater than the pivot character. We swap the pivot and the successor, and then sort the suffix to the right of the pivot in increasing order to get the next lexicographically greater string.

We then return the resulting string. We change string to a list using list function so that we can modify it in place by swapping characters and sorting the suffix. Finally, we convert the list into the string by join function.

# **Exercise 3: ⬤**

omberman lives in a rectangular grid. Each cell in the grid either contains a omb or nothing at all.

Each omb can be planted in any cell of the grid but once planted, it will detonate after exactly 3 seconds. Once a omb detonates, it's destroyed — along with anything in its four neighboring cells. This means that if a omb detonates in cell *i, j*, any valid cells ( i ± 1, j ) and ( i, j ± 1 ) are cleared. If there is a omb in a neighboring cell, the neighboring omb is destroyed without detonating, so there's no chain reaction.

omberman is immune to ombs, so he can move freely throughout the grid. Here's what he does:

1. Initially, omberman arbitrarily plants ombs in some of the cells, the initial state.

2. After one second, omberman does nothing.

3. After one more second, omberman plants bombs in all cells without ombs, thus filling the whole grid with ombs. No ombs detonate at this point.

4. After one more second, any ombs planted exactly three seconds ago will detonate.

Here, omberman stands back and observes.

5. omberman then repeats steps 3 and 4 indefinitely.

Note that during every second omberman plants ombs, the ombs are planted simultaneously (i.e., at the exact same moment), and any ombs planted at the same time will detonate at the same time.

Given the initial configuration of the grid with the locations of omberman's first batch of planted ombs, determine the state of the grid after **N** seconds.

For example, if the initial grid looks like:

. . .

.O.

. . .

It looks the same after the first second. After the second second, omberman has placed all his charges:

OOO

OOO

OOO

At the third second, the bomb in the middle blows up, emptying all surrounding cells:

O.O

...

O.O

## **Function Description**

Create the omber\_man function with following parameter(s):

● int n: the number of seconds to simulate

● string grid[r]: an array of string that represents the grid

## **Returns**

● string[r]: n array of string that represent the grid in its final state

**Sample Input:**

3

. . . . . . .

. . . O . . .

. . . . O . .

. . . . . . .

OO . . . . .

OO . . . . .

## **Sample Output**

OOO . OOO

OO . . . OO

OOO . . . O

. . OO . OO

. . . OOOO

. . . OOOO

Project:

Python code:

def bomber\_man(n, grid):

grid = [list(row) for row in grid]

rows, cols = len(grid), len(grid[0])

def detonate(positions):

for r, c in positions:

grid[r][c] = '.'

if r > 0 and grid[r-1][c] == 'O':

grid[r-1][c] = '.'

if r < rows-1 and grid[r+1][c] == 'O':

grid[r+1][c] = '.'

if c > 0 and grid[r][c-1] == 'O':

grid[r][c-1] = '.'

if c < cols-1 and grid[r][c+1] == 'O':

grid[r][c+1] = '.'

if n == 1:

return grid

elif n == 2:

return [['O' for \_ in range(cols)] for \_ in range(rows)]

detonate([])

if n == 3:

return grid

previous\_positions = set()

current\_positions = set()

for i in range(4, n+1):

detonate(previous\_positions)

for r in range(rows):

for c in range(cols):

if grid[r][c] == '.':

grid[r][c] = 'O'

current\_positions.add((r, c))

previous\_positions = current\_positions

current\_positions = set()

return grid

there are two main arguments, an integer n representing and list of strings grid representing the initial configuration of the grid. Each string in the list represents a row of the grid, with “.” Represents denoting an empty cell and “O” denoting a bomb.

The function first transfersthe grid into a list of lists, since lists are easier to use in this case. It then defines a helper function detonate that takes a set of positions and detonates the bombs at those positions and their neighboring cells.

Then there are special cases where n is 1, 2, or 3. If n is 1 or 2, the function returns the initial grid or an all-bombs grid, respectively. If n is 3, the function returns the grid after the bombs planted at the second second are detonated.

For n greater than 3, the function makes simulation of the behavior of Bomberman by alternating between planting bombs and detonating them every two seconds. It uses two sets of positions to keep track of the bombs to be detonated in the previous and current rounds.In the end function returns the resulting grid after n seconds. Note that the function modifies the grid in place, so if you want to preserve the original grid, you should make a copy of it before calling the function.

## **REST URL Shortener API**

URL shortening is a technique on the World Wide Web in which a Uniform Resource Locator (URL) may be made substantially shorter and still direct to the required page. This is achieved by using a redirect which links to the web page that has a long URL. For example, the URL "https://example.com/assets/category\_B/subcategory\_C/Foo/" can be shortened to

"https://example.com/Foo", and the URL "https://en.wikipedia.org/wiki/URL\_shortening" can be shortened to "https://w.wiki/U". Often the redirect domain name is shorter than the original one. A friendly URL may be desired for messaging technologies that limit the number of characters in a message (for example SMS), for reducing the amount of typing required if the reader is copying a URL from a print source, for making it easier for a person to remember, or for the intention of a permalink. In November 2009, the shortened links of the URL shortening service Bitly were accessed 2.1 billion times.

Other uses of URL shortening are to "beautify" a link, track clicks, or disguise the underlying address. Although disguising of the underlying address may be desired for legitimate business or personal reasons, it is open to abuse.[2] Some URL shortening service providers have found themselves on spam blocklists, because of the use of their redirect services by sites trying to bypass those very same blocklists. Some websites prevent short, redirected URLs from being posted.

Required features:

- API to create URL with random short name (something like this url/78sda8s6d), url random name should be fixed size string and unique per url

- API for premium clients to create URL with custom name (url/<custom>)

- Input url validation, that it is a correct url and size must be below 250 characterAu

- Counters - how many times the url was accessed (optional requirement)

- Automatic deletion of urls older than 30 days (optional requirement)

In my opinion In Order to implement the required features above, we can create a URL shortening service with the following ways:

Database: We firs create Database for storing purposes,We can aither use a relational database like MySQL or a NoSQL database like MongoDB.

API: We would also need to create APIs to handle the following components.

a. Create a short URL with a random name

b. Create a short URL with a custom name: This API should allow premium clients to specify a custom name for the short URL. It should make sure that the custom name is not already used and store it along with the original URL in the database.

c. Redirect to the original URL: This API should take a short URL as input, look it up in the database, and redirect the user to the original URL.

Input validation: We need to validate the input URL to ensure that it is a valid URL and its size is below 250 characters. We can use regular expressions to validate the URL.

Counters: We can store a counter for each URL to track how many times it was accessed. We can increment the counter each time the URL is accessed and store the updated count in the database.

Automatic deletion: We can add a job to the server that runs daily to delete URLs that are older than 30 days from the database.