Задача A. Без мата не обошлось...

Имя входного файла:	`kingrook.in`
Имя выходного файла:	`kingrook.out`
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мебибайт

Вам дана шахматная доска размера 6×6, на которой стоят три шахматные фигуры: белый король, белая ладья и чёрный король. Ваша задача — рассчитать минимальное число ходов, требуемое белым для того, чтобы заматовать чёрного короля, или определить, что это невозможно, либо что позиция является некорректной.

Формат входных данных

Во входном файле одна строка, в которой записаны три координаты полей — координаты белого короля, белой ладьи и чёрного короля соответственно. После координат через пробел написан идентификатор стороны, которая делает первый ход (‘`W`’, если белые, или ‘`B`’, если чёрные).

Формат выходных данных

Выведите в выходной файл общее количество ходов обеих сторон, необходимое для белых, чтобы выиграть игру. Если чёрный король заматован, выведите `0`. Если входная позиция некорректна, выведите `-1`. Если игра закончиться вничью (например, на доске пат), выведите `-2`.

Примеры

kingrook.in	kingrook.out
c6 f4 a5 B	2
c6 f4 b5 W	-1

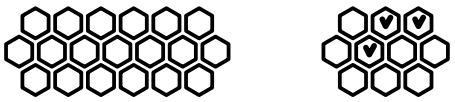
Задача B. Игра с шестиугольниками

Имя входного файла:	`hexgame.in`
Имя выходного файла:	`hexgame.out`
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мебибайт

Рассмотрим полоску высоты 3, состоящую из правильных шестиугольников. Верхний и нижний ряды полоски состоят из n шестиугольников, а средний ряд содержит $n + 1$ шестиугольник.

Два игрока по очереди ставят одинаковые отметки в шестиугольники. В каждом шестиугольнике может стоять не более одной отметки. Выигрывает тот, после чьего хода оказываются отмечены три шестиугольника такие, что один из них — соседний по стороне с двумя другими, а два отрезка, проведённые из его центра в центры этих двух соседей, образуют угол в 120 градусов.

На рисунке слева показан пример полоски для $n = 6$, а на рисунке справа — пример трёх отметок, образующих выигрышную комбинацию шестиугольников.



Дана позиция в этой игре, в которой никто ещё не выиграл. Кто выигрывает при правильной игре, начинающейся с этой позиции: первый игрок или второй? Первым игроком считается тот, кто делает из данной позиции первый ход. Выражение «при правильной игре» можно определить рекурсивно: если игрок может сделать такой ход, чтобы либо сразу выиграть, либо после любого ответного хода противника по-прежнему иметь возможность выиграть при правильной игре, он обязательно сделает один из таких ходов. Если же такого хода не существует, игрок может сделать любой из возможных ходов.

Формат входных данных

В первой строке входного файла задано целое число t — количество тестов ($1 \leqslant t \leqslant 10$). Далее записаны сами тесты. Каждый тест представляет собой позицию в игре и задаётся четырьмя строками. В первой из этих строк записано целое число n — количество шестиугольников в верхнем и нижнем рядах полоски. ($1 \leqslant n \leqslant 5000$). Напомним, что в таком случае средний ряд содержит $n + 1$ шестиугольник. Следующие три строки задают сами полоски: верхнюю, нижнюю и среднюю. Символ «`.`» (точка) соответствует пустому шестиугольнику, а «`#`» (решётка) — отмеченному. Гарантируется, что в заданной позиции никто ещё не выиграл.

Формат выходных данных

В ответ на каждый из t тестов выведите на отдельной строке слово «`FIRST`», если из позиции, заданной в этом тесте, выигрывает первый игрок, или же слово «`SECOND`», если из этой позиции выигрывает второй игрок.

Пример

hexgame.in	hexgame.out	иллюстрация
3 2 .. .. 3 #.# ...# ##. 2 .. .#. ..	FIRST FIRST SECOND	

Пояснение к примеру

В первом тесте первый игрок может поставить отметку в центральный шестиугольник и заставить второго играть с позиции, представленной в третьем тесте.

Во втором тесте у первого игрока существует несколько способов выиграть первым же ходом.

В третьем тесте в ответ на любой из шести ходов существует два различных хода, несущих победу второму игроку.

Задача C. Дискретное логарифмирование

Имя входного файла:	`log.in`
Имя выходного файла:	`log.out`
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мебибайт

Даны натуральные числа a, b, n. Требуется найти *дискретный логарифм* b по основанию a по модулю n, то есть такое число x ($0 \leqslant x < n$), что $a^x \equiv b \pmod{n}$.

Формат входных данных

В первой строке заданы через пробел три целых числа a, b и n ($1 \leqslant a, b, n \leqslant 10^{12}$).

Формат выходных данных

В первой строке выходного файла выведите -1, если дискретного логарифма не существует. Иначе следует вывести его значение.

Если ответ неоднозначен, разрешается выводить любой.

Пример

log.in	log.out
2 4 6	2

Задача D. Универсальный замедлитель быстрой сортировки

Ограничение по времени: 5 секунд
Ограничение по памяти: 256 мебибайт

Внимание! Эту задачу можно сдать только со следующими языками и компиляторами: GNU C++ 5.1.0 (TDM-GCC-32) для языка C++, GNU C 5.1.0 (TDM-GCC-32) для языка C, Free Pascal 2.6.4 для языка Pascal.

Вы, наверное, слышали о существовании QuickSort — алгоритма быстрой сортировки. Этот алгоритм сортирует данный ему массив объектов a_0, a_1, …, a_{n-1}, на которых определено отношение порядка:

- Для любых двух объектов a_i и a_j известно, что либо $a_i < a_j$, либо $a_i = a_j$, либо $a_i > a_j$.
- Каждый объект равен самому себе: $a_i = a_i$.
- Выполнено условие *транзитивности*: если $a_i < a_j$ и $a_j < a_k$, то $a_i < a_k$.

В этой задаче мы будем рассматривать конкретный класс реализаций этого алгоритма, который подробно описан ниже. Во всей задаче размер массива $n = 10\,000$. Реализация, с которой мы будем иметь дело, состоит из двух функций: `quicksort` и `select_pivot`.

Функция `quicksort` получает в качестве аргументов два числа l и r, означающих, что нужно отсортировать отрезок массива с индексами от l до r включительно, а также функцию сравнения `cmp`. Сначала функция проверяет, что $r > l$: в противном случае отрезок уже отсортирован. Затем вызывается функция `select_pivot` с таким же набором аргументов, которая каким-то образом выбирает один элемент на отрезке в качестве разделителя. Далее `quicksort` помещает все элементы, меньшие выбранного, слева от него, все бóльшие — справа, а все равные ему, включая сам разделитель, оказываются в середине. В процессе этих перемещений отрезок делится на две меньшие части такие, что любой элемент в левой части не больше разделителя, а любой элемент в правой части не меньше разделителя; возможно, между этими частями остаётся один элемент, равный разделителю. Наконец, `quicksort` вызывает сама себя от двух полученных частей. Используя метод математической индукции, можно доказать, что после вызова `quicksort (0, n - 1, cmp)` отрезок оказывается отсортирован.

Известно, что многие реализации алгоритма QuickSort обладают существенным недостатком: если известны детали реализации, можно научиться строить такой массив длины n, что время работы алгоритма будет иметь порядок $O(n^2)$. Чуть менее известно, что можно строить такой массив, даже если детали реализации скрыты, но верны некоторые общие предположения о них.

В этой задаче вам предстоит строить массивы, на которых реализации QuickSort, придуманные жюри, работают достаточно долго. Эти реализации не имеют прямого доступа к массиву a. Для сортировки они вместо изменения массива строят вспомогательную перестановку p. В начале работы алгоритма перестановка тождественная: $p_0 = 0$, $p_1 = 1$, …, $p_{n-1} = n - 1$. После выполнения алгоритма массив отсортирован в следующем смысле: $a_{p_0} \leqslant a_{p_1} \leqslant a_{p_2} \leqslant \ldots \leqslant a_{p_{n-1}}$. Заметим, что при такой реализации отрезок массива от l до r, с которым работает функция `quicksort`, состоит из элементов a_{p_l}, $a_{p_{l+1}}$, …, a_{p_r}, которые в самом массиве a не обязательно идут подряд.

Каждый раз, когда нужно сравнить два элемента массива с индексами i и j, вызывается функция сравнения `cmp (p[i], p[j])`. Временем работы алгоритма будем считать количество вызовов функции сравнения.

Одна часть реализации алгоритма — функция `quicksort` — открыта и доступна для ознакомления. Ниже приведён псевдокод этой функции. Реализация на языках программирования C/C++ и Pascal доступна в грейдерах, выданных участникам.

```
quicksort (l, r, cmp):
    if l >= r:
        exit
    pivot = p[select_pivot (l, r, cmp)]
    i = l
    j = r
    while i <= j:
        while cmp (p[i], pivot) < 0:
            i += 1
        while cmp (pivot, p[j]) < 0:
            j -= 1
        if i <= j:
            p[i] <--> p[j]
            i += 1
            j -= 1
    quicksort (l, j, cmp)
    quicksort (i, r, cmp)
```

Другая часть реализации — функция `select_pivot` — своя в каждом тесте к этой задаче. В первом тесте эта функция всегда выбирает первый элемент отрезка в качестве разделителя. Во всех остальных тестах реализация недоступна участникам; она может произвести несколько сравнений и перестановок и даже пользоваться генератором случайных чисел. Псевдокод этой функции для первого теста приведён ниже. Реализация на языках программирования C/C++ и Pascal также доступна в грейдерах, выданных участникам.

```
select_pivot (l, r, cmp):
    return l
```

Условия, выполняющиеся для функции `select_pivot`, таковы:

- Эта функция, как и `quicksort`, не имеет прямого доступа к массиву a.
- Каждый раз, когда ей нужно сравнить два элемента массива с индексами i и j, она вызывает функцию `cmp (p[i], p[j])`.
- Гарантируется, что при каждом вызове сравниваются два элемента, принадлежащие текущему отрезку массива от l до r, то есть $l \leqslant i, j \leqslant r$. Заметим, что, поскольку алгоритм поддерживает внутри себя перестановку p для элементов массива a, аргументы функции `cmp` не обязательно лежат в пределах от l до r.

Следующие два условия формализуют для целей нашей задачи общее требование того, чтобы функция `select_pivot` выполняла не больше $O(1)$ действий, то есть не могла существенно изменить поведение алгоритма QuickSort.

- При каждом вызове `select_pivot` вызывает `cmp` **не более трёх раз**.
- При каждом вызове `select_pivot` переставляет два элемента в массиве p с индексами от l до r **не более трёх раз**. Никаких других изменений в массив p эта функция вносить не может.

Вспомогательную часть реализации — функцию `cmp` — пишете вы. Сам массив a должен быть предоставлен для проверки в конце работы алгоритма QuickSort. До этого момента вам требуется всего лишь при каждом вызове `cmp (i, j)` отвечать, верно ли, что $a_i < a_j$ (тогда следует вернуть любое отрицательное число), или $a_i = a_j$ (следует вернуть ноль), или же $a_i > a_j$ (следует вернуть положительное число). При этом вы, конечно, можете ответить что-нибудь, даже если ещё не решили, чему будут в итоге равны эти элементы. Как и в какой момент заполнять сами элементы массива a — ваше дело.

Если ваша функция выдаёт не совместимые друг с другом результаты (например, $a_0 < a_1$ и $a_1 < a_0$), это может привести к тому, что корректная работа QuickSort будет нарушена. В таком случае работа программы будет прекращена, а ваше решение получит вердикт `Wrong Answer` на соответствующем тесте.

После окончания работы алгоритма необходимо предоставить массив a для проверки. Алгоритм QuickSort будет запущен ещё раз, но уже для этого массива: ваша функция сравнения

уже не будет запускаться, вместо этого будут просто сравниваться два соответствующих элемента.

Ваше решение будет считаться правильным, если на массиве, предоставленном для проверки, функция сравнения будет вызвана хотя бы $3\,000\,000$ раз. Гарантируется, что реализация `select_pivot` во время проверки будет вести себя точно так же, как и до проверки: например, генератор случайных чисел, если он используется, будет в начале работы проинициализирован тем же значением. В частности, для любого k верно следующее: если первые k сравнений выдали такие же ответы при проверке, как и во время первого запуска алгоритма, то перестановка p находится в том же состоянии, а $(k+1)$-е сравнение при проверке будет вызвано с теми же аргументами, что и $(k+1)$-е сравнение во время первого запуска алгоритма.

Заметьте, что формально не проверяется соответствие поведения вашей функции `cmp` во время первого запуска и поведения обычного сравнения во время проверочного запуска. Тем не менее, первое же несоответствие фактически аннулирует гарантию предсказуемости дальнейшего поведения.

Напишите программу, которая решает поставленную задачу. Для этого реализуйте следующие три функции.

Функция `init (n)` вызывается один раз в начале работы программы. Она получает в качестве аргумента одно число — длину массива. Напомним, что во всех вызовах $n = 10\,000$.

Функция `cmp (i, j)` вызывается не известное заранее количество раз из функций `quicksort` и `select_pivot`. Она должна вернуть отрицательное число, если $a_i < a_j$, ноль, если $a_i = a_j$, и положительное число, если $a_i > a_j$. Результатом может быть любое 32-битное целое число со знаком.

Функция `fill_array (a)` вызывается один раз после всех вызовов остальных функций. Она должна заполнить проверочный массив длины n элементами так, чтобы при проверке количество сравнений оказалось не меньше $3\,000\,000$. Каждый элемент массива может быть любым 32-битным целым числом со знаком.

Всего в этой задаче 10 тестов.

Задача E. Порядок циклов

Имя входного файла:	`order.in`
Имя выходного файла:	`order.out`
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мебибайт

Дан неориентированный граф из n вершин, заданный матрицей смежности `a` (`a[u][u] = True`; `a[u][v] = a[v][u]`; `a[u][v] = True` тогда и только тогда, когда есть ребро между вершинами `u` и `v`). На нём запускают следующий алгоритм:

```
for x := 1 to n do
  for y := 1 to n do
    for z := 1 to n do
      if a[i][k] and a[k][j] then
        a[i][j] := True;
```

Перед запуском буквы `x`, `y` и `z` заменяют буквами `i`, `j` и `k` в некотором порядке. Утверждается, что после работы этого алгоритма `a[u][v] = True` тогда и только тогда, когда в исходном графе существует путь между вершинами `u` и `v`. Выясните, верно ли это, и если нет, приведите пример исходного графа, на котором это неверно.

Формат входных данных

В первой строке ввода записаны через пробел три буквы — '`i`', '`j`' и '`k`' — в некотором порядке. Первая буква подставляется в программу вместо '`x`', вторая — вместо '`y`', третья — вместо '`z`'.

Формат выходных данных

Если искомый граф существует, в первой строке выведите через пробел целые числа n и m — количество вершин и рёбер в графе, соответственно ($1 \leqslant n \leqslant 10$, $0 \leqslant m \leqslant 45$). В следующих m строках выведите пары вершин, соединённых рёбрами, по одной паре на строке. Номера вершин в паре должны быть упорядочены по возрастанию; вершины нумеруются с единицы. Кратные рёбра и петли не допускаются.

Если же программа с заданным порядком циклов корректно работает на любом графе, вместо n и m выведите в первой строке два нуля через пробел.

Пример

order.in	order.out
k i j	0 0

Задача F. Векторы в реальном времени

Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мебибайт

Внимание! Эту задачу можно сдать только со следующими языками и компиляторами: GNU C++ 5.1.0 (TDM-GCC-32) для языка C++, GNU C 5.1.0 (TDM-GCC-32) для языка C, Free Pascal 2.6.4 для языка Pascal.

Вы, наверное, слышали о существовании структуры данных `std::vector` (вектор) в библиотеке STL для C++. Эта структура данных представляет собой массив с возможностью доступа за $O(1)$ и изменения длины, а также добавления в конец. Вектор, состоящий из n элементов, занимает $O(n)$ памяти. Известно, что суммарное время, которое потратится на добавление n элементов в конец изначально пустого вектора, равно $O(n)$. Тем не менее, каждое отдельное добавление может само по себе занять $O(n)$ времени. О таком поведении структуры данных говорят, что добавление элемента работает в среднем за $O(1)$, или, как ещё говорят, за амортизационную стоимость $O(1)$.

В этой задаче требуется реализовать схожую структуру данных, реализующую массив изменяемой длины, на которую накладываются следующие ограничения:

- Структура данных должна поддерживать операции добавления в конец массива и выдачи элемента массива по его номеру.
- Хранение n элементов должно требовать $O(n)$ памяти и использовать $O(1)$ различных объектов, предоставляемых менеджером памяти.
- Выполнение каждой операции должно производить $O(1)$ элементарных действий в худшем случае.

Будем называть эту структуру real-time vector или, для краткости, просто вектором.

Проверочную программу, которая должна получиться в результате решения задачи, можно условно разделить на три уровня: верхний, средний и нижний. При работе программы верхний уровень будет вызывать функции среднего уровня, а они, в свою очередь, должны пользоваться функциями нижнего уровня.

Нижний уровень программы предоставляет жюри: это эмуляция менеджмента памяти на уровне системы. Этот уровень состоит из пяти библиотечных функций, однократный вызов каждой из которых считается одним элементарным действием. Любая ошибка, возникающая при вызове этих функций, немедленно приводит к вердикту `Wrong Answer`.

- Функция `lib_allocate (size)` выделяет блок памяти под массив из `size` элементов. Она возвращает целое число — условный идентификационный номер этого массива, которым можно впоследствии пользоваться для доступа к нему.
- Функция `lib_release (array_id)` освобождает блок памяти, занятый массивом с номером `array_id`. Если память под массив с таким номером не выделена на момент вызова этой функции, программа завершает работу с ошибкой.
- Функция `lib_get (array, index)` выдаёт для проверки на верхнем уровне элемент массива с номером `array`, имеющий индекс `index`. Все массивы индексируются с нуля. Если при вызове этой или следующих функций массива с таким номером или элемента с таким индексом не существует, программа завершает работу с ошибкой. Чтобы лучше понять, как пользоваться этой функцией, прочитайте описание функции среднего уровня `vec_get` ниже.
- Функция `lib_copy (dest_array, dest_index, src_array, src_index)` копирует элемент массива с номером `src_array`, имеющий индекс `src_index`, поверх элемента массива с номером `dest_array`, имеющий индекс `dest_index`.

- Функция `lib_put (dest_array, dest_index)` копирует элемент, заданный на верхнем уровне проверки, поверх элемента массива с номером `dest_array`, имеющего индекс `dest_index`. Чтобы лучше понять, как пользоваться этой функцией, прочитайте описание функции среднего уровня `vec_push` ниже.

Средний уровень программы пишет участник: это реализация одного экземпляра real-time vector на основе элементарных действий нижнего уровня. Каждая функция этого уровня должна выполнять $O(1)$ элементарных действий. Заметим, что количество действий, не использующих менеджер памяти, а также размер дополнительно используемой памяти, ограничены только ограничениями по времени и памяти в задаче. Этот уровень состоит из трёх функций.

- Функция `vec_init ()` вызывается один раз в начале работы программы. Она может произвести какие-то подготовительные действия.
- Функция `vec_push ()` должна добавить элемент в конец вектора. Заметьте, что в целях проверки сам элемент x не предоставляется среднему уровню программы, контролируемому участником. Чтобы сохранить этот элемент для последующего использования, следует один или более раз выполнить элементарное действие `lib_put` нижнего уровня, которому будет неявно предоставлен элемент x. Если в момент вызова этой функции длина вектора считается равной k, то после него длина должна считаться равной $k + 1$, а элементом с номером k должен считаться элемент x.
- Функция `vec_get (index)` должна возвратить элемент, который считается находящимся в векторе на позиции `index`. Элементы вектора нумеруются по порядку, начиная с нулевого. Гарантируется, что в момент вызова этой функции вектор уже должен содержать элемент с номером `index`. Заметьте, что в целях проверки сам элемент, который необходимо выдать на проверку, недоступен среднему уровню программы, контролируемому участником. Вместо этого следует ровно один раз выполнить элементарное действие нижнего уровня `lib_get (array, index)` с такими параметрами, чтобы менеджер памяти выдал верхнему уровню нужный элемент. Если неверно, что выдан правильный элемент и выдача элемента произошла ровно один раз за время вызова функции `vec_get`, программа завершает работу с вердиктом `Wrong Answer`.

Верхний уровень программы отвечает за общий ход проверки, вызывая функции среднего уровня и проверяя результаты их работы. Этот уровень сначала вызывает один раз функцию `vec_init`, а затем производит суммарно от 10 до 1 000 000 вызовов двух других функций среднего уровня. Точное количество и порядок этих вызовов зафиксированы отдельно в каждом тесте. Гарантируется, что первым после `vec_init` идёт вызов `vec_push`. Параметр функции `vec_get` каждый раз выбирается псевдослучайным образом: независимо и равномерно среди всех корректных на этот момент значений, но при этом одинаково при повторных запусках одного и того же решения на одном и том же тесте.

Реализуйте функции среднего уровня так, чтобы решить поставленную задачу. Изначально считается, что вектор пуст.

Всего в этой задаче один пример, доступный в архиве с грейдерами, и 10 основных тестов.

В течение проверки верхний уровень программы отслеживает максимальное значение трёх величин. Первая из них — отношение суммарной длины массивов, выделенных и ещё не освобождённых менеджером памяти, к сумме единицы и количества элементов, которые в этот момент должны содержаться в векторе. Вторая — количество вызовов элементарных действий во время каждого вызова каждой из трёх функций, реализованных участником. Третья — количество выделенных и не освобождённых массивов менеджера памяти в каждый момент времени. Все эти величины по сути задачи ограничены оценкой $O(1)$. В целях проверки считается, что оценка для каждой из величин в отдельности достигнута, если эта величина не превосходит константы 5.

Напомним, что, если участник неправильно обработал какой-то вызов функции среднего уровня, решение получает за соответствующий тест 0 баллов с вердиктом `Wrong Answer`.

Задача G. Сравнение строк

Имя входного файла:	`compare.in`
Имя выходного файла:	`compare.out`
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мебибайт

Циклическое расширение $\mathcal{S}^*$ строки $\mathcal{S}$ — это строка $\mathcal{S}$, приписанная сама к себе бесконечное количество раз; к примеру, циклическим расширением строки «`bab`» является строка «`babbabbab...`».

Даны длины двух строк $\mathcal{S}$ и $\mathcal{T}$ — числа m и n. Рассмотрим циклические расширения $\mathcal{S}^*$ и $\mathcal{T}^*$ этих строк. Как проверить, равны ли они?

Наивный алгоритм будет проверять строки на равенство, просто сравнивая s_1 с t_1, s_2 с t_2 и так далее. В итоге алгоритм либо найдёт пару несовпадающих символов, либо, если строки равны, будет проводить сравнения бесконечно долго.

Однако понятно, что, если строки не равны, то последняя позиция p, на которой мы можем встретить различие ($s_p \neq t_p$), конечна. Мы хотим узнать, чему равно p, чтобы улучшить алгоритм сравнения так: новый алгоритм будет сравнивать символ s_1 с t_1, s_2 с t_2 и так далее, пока либо не найдёт несовпадение $s_q \neq t_q$ на позиции $q \leqslant p$, либо, сравнив первые p пар и обнаружив соответствие символов в каждой паре, не докажет тем самым равенство строк $\mathcal{S}^*$ и $\mathcal{T}^*$.

Для данных длин строк m и n приведите пример строк $\mathcal{S}$ и $\mathcal{T}$ соответствующей длины, циклические расширения которых различны, но первое несовпадение встречается как можно позже.

Формат входных данных

В первой строке ввода заданы два числа через пробел — это числа m и n ($1 \leqslant m, n \leqslant 100$).

Формат выходных данных

Выведите в первой строке строку $\mathcal{S}$ из m символов, а во второй — строку $\mathcal{T}$ из n символов. Строки могут содержать только маленькие буквы английского алфавита (`'a'`–`'z'`).

Пример

compare.in	compare.out
2 4	aa aaab

Пояснение к примеру

Для $m = 2$ и $n = 4$ значение p равно четырём, и оно достигается на строках «`aa`» и «`aaab`», циклические расширения которых — строки «`aaaa...`» и «`aaab...`» — различаются в четвёртом символе.

Задача H. Слова

Имя входного файла:	`words.in`
Имя выходного файла:	`words.out`
Ограничение по времени:	2 секунды
Ограничение по памяти:	256 мебибайт

Дан набор из n различных слов. Для каждого слова узнайте, сколько раз оно встречается как подстрока во всех остальных словах.

Формат входных данных

В первой строке записано целое число n ($1 \leqslant n \leqslant 10\,000$). В следующих n строках записаны слова. Каждое слово непусто и состоит из не более чем 20 строчных букв латинского алфавита. Все слова различны.

Формат выходных данных

Выведите n строк, по одному числу на строке. В i-й строке должно быть записано, сколько раз i-е слово встречается в других словах как подстрока.

Примеры

words.in	words.out
1 word	0
2 aba abacaba	2 0
5 less lesss session s ss	1 0 0 10 4