Задача 1-1 (50 баллов). В первой строке входного потока записано число n. Во второй строке записаны n ($1 \le n \le 1000$) целых чисел $a_1, a_2, a_3, \ldots, a_n$ ($|a_i| \le 10^9$). Найдите наибольшую чередующуюся подпоследовательность $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ последовательности $a_1, a_2, \ldots, a_n$, то есть такую подпоследовательность, для которой $i_1 < i_2 < \ldots < i_k$, любые два соседних элемента различны, и для любых трех соседних элементов $a_{i_{l-1}}, a_{i_l}, a_{i_{l+1}}$ либо $a_{i_{l-1}} < a_{i_l}, a_{i_l} > a_{i_{l+1}}$, либо $a_{i_{l-1}} > a_{i_l}, a_{i_l} < a_{i_{l+1}}$, при этом k — наибольшее возможное. В выходной поток выведите саму подпоследовательность $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$. Если таких последовательностей несколько, то следует выбрать ту, для которой i_1 минимально. Из всех максимальных с одинаковым i_1 — ту, у которой значение i_2 минимально и так далее.

Пример входа	Пример выхода
10 1 4 2 3 5 8 6 7 9 10	1 4 2 8 6 7
5 1 2 3 4 5	1 2
1 100	100

Решение.

Эту задачу предполагалось решать алгоритмом, аналогичным рассказанному на семинаре алгоритму за $O(n^2)$ для решения задачи LIS. В отличие от LIS в этой задаче есть не один тип подпоследовательности — возрастающая подпоследовательность — а два типа: чередующаяся, начинающаяся с возрастания (Up), и чередующаяся, начинающаяся с убывания (Down). Поэтому нам придется вычислить два массива значений: наибольшие длины чередующихся последовательностей обоих типов, начинающихся в позиции i, для каждого i от 0 до $n-1$. Почему мы будем считать именно наибольшие длины подпоследовательностей, начинающихся в позиции i, а не заканчивающихся в позиции i? Потому что так правильно делать в любой задаче, в которой требуется найти лексикографически наименьший в смысле последовательности индексов ответ.

Действительно, если мы уже знаем длины последовательностей, начинающихся в каждой позиции, то нам очень легко искать сразу лексикографически наименьший ответ. Чтобы ответ получился наименьшим лексикографически, необходимо все время пытаться брать в нашу подпоследовательность элементы с как можно меньшими индексами: сначала попытаться взять самый первый элемент, если мы уверены, что дальше получится добавить к нему несколько элементов, чтобы получить самую длинную чередующуюся подпоследовательность. После того, как мы определились, брать или не брать первый элемент, нужно попытаться взять второй, потом третий и т.д., пытаться брать каждый элемент. Мы можем взять элемент i, если выполнены следующие условия (но сначала введем несколько обозначений):

Обозначим нашу последовательность за A, а строящийся постепенно ответ (подпоследовательность в A) — за X. Элементы в A и в X нумеруются с нуля, при этом в A всегда всего n элементов, в X - всего k элементов в данный момент. Обозначим $MaxUp[i]$ — длина наибольшей чередующейся подпоследовательности, начинающейся в i-м элементе, у которой второй элемент больше первого. Обозначим $MaxDown[i]$ — длина аналогичной подпоследовательности, у которой второй элемент меньше первого. Также может быть,

что $MaxUp[i] = 1$ или $MaxDown[i] = 1$, тогда второго элемента у соответствующей подпоследовательности нет, и она является тривиальной чередующейся подпоследовательностью. Обозначим также $L = \max\limits_{i=0,\ldots,n-1}(MaxUp[i], MaxDown[i])$ — длина ответа к задаче.

Итак, i-й элемент добавляется к ответу (к X), если выполнены два условия:

1. i-й элемент подходит к уже построенному начальному куску подпоследовательности. Т.е. если $k \geq 2$ и $X[k-2] > X[k-1]$, то должно выполняться $A[i] > X[k-1]$, чтобы i-й элемент можно было взять, а если $X[k-2] < X[k-1]$, то и $A[i]$ должно быть меньше, чем $X[k-1]$, иначе подпоследовательность не получилась бы чередующейся. Если $k = 1$, то должно выполняться $A[i] \neq X[0]$, иначе подпоследовательность получилась бы нестрого чередующейся.

2. Если $k \geq 1$: если $X[i] > X[k-1]$, то должно быть $L = k + MaxDown[i]$, а если $X[i] < X[k-1]$, то должно быть $L = k + MaxUp[i]$. Потому что подпоследовательность, начинающаяся с i-го элемента должна будет быть соответственно типа $Down$ или типа Up, при этом общая длина ответа будет состоять из длины уже построенного куска X и длины этой подпоследовательности, и она должна совпадать с максимально возможной длиной. Если же $k = 0$, то должно быть $L = \max(MaxUp[i], MaxDown[i])$.

Таким образом, если известны все значения $MaxDown[i]$ и $MaxUp[i]$, то мы простым линейным проходом по массиву составляем лексикографически наименьший ответ.

Как же найти все эти значения? Динамическим программированием, как и ответы к задаче LIS. Чтобы найти $MaxDown[i]$ и $MaxUp[i]$, нам необходимо будет знать все значения $MaxDown[j]$ и $MaxUp[j]$ для всех $j > i$. Поэтому мы будем вычислять оба этих массива с конца. Для $i = n-1$ очевидно, что $MaxDown[i] = MaxUp[i] = 1$. Теперь будем уменьшать i по единице и искать ответ:

$MaxDown[i] = \max\limits_{j=i+1,\ldots,n-1;A[j]<A[i]}(MaxUp[j] + 1)$,

$MaxUp[i] = \max\limits_{j=i+1,\ldots,n-1;A[j]>A[i]}(MaxDown[j] + 1)$.

Каждый шаг такого алгоритма выполняется за $O(n)$, всего шагов n, поэтому сложность получается $O(n^2)$. Затраты памяти линейны.

Оказалось, однако, что эту задачу можно решить и быстрее, за $O(n)$. Для этого заметим во-первых, что все подпоследовательности из подряд идущих одинаковых элементов можно заменить на один такой элемент, и от этого ответ к задаче не изменится. После этого вся последовательность однозначно разбивается на отрезки строго возрастания и убывания. При этом очевидно, что отрезки возрастания и убывания чередуются. Т.е. если $A[1] > A[0]$, то сначала несколько элементов подряд строго возрастают, потом несколько элементов подряд строго убывают, потом опять возрастают, убывают и т.д., пока мы не дойдем до конца последовательности. Если же $A[1] < A[0]$, то все с точностью до наоборот. Не может быть $A[1] = A[0]$, и не может быть вообще никаких двух последовательных одинаковых элементов, т.к. мы все непрерывные последовательности из одинаковых элементов уже "сжали" до одного элемента.

Можно доказать, что ответ к задаче L равен количеству этих отрезков возрастания-убывания плюс единица. Очевидно, что чередующуюся подпоследовательность такой

длины можно построить: для этого достаточно взять в нее начало и конец последовательности, а так же все элементы, в которых заканчивается один отрезок и начинается другой. То, что подпоследовательность длиннее построить нельзя, оставляется в качестве упражнения. Указанную подпоследовательность легко построить за время $O(n)$.

Указанная подпоследовательность не будет обязательно являться лексикографически наименьшей, однако из нее можно получить лексикографически наименьшую: будем по очереди пытаться сдвигать все элементы текущей подпоследовательности налево. Будем без ограничения общности считать, что второй элемент оптимальной последовательности больше первого. Самый первый сдвигать влево некуда. Второй можно сдвигать влево до тех пор, пока он остается больше следующего. Как только мы его таким образом максимально сдвинем влево, уже нельзя будет его сдвинуть влево в будущем, т.к. следующий элемент мы уменьшить никак не можем, двигая его влево или вправо по своим двум отрезкам. Сдвигать же следующий элемент за пределы своего отрезка означало бы уменьшить максимально возможную длину всей последовательности. Соответственно, мы сдвигаем сначала второй элемент как можно левее, затем третий элемент как можно левее, но так, чтобы он был меньше и второго, и четвертого. Затем четвертый элемент сдвигаем как можно левее, но так, чтобы он оставался больше третьего и пятого. И т.д. Здесь нет строгого доказательства, что нельзя добиться лучшей подпоследовательности, чем полученная таким образом, это доказательство также оставляется в качестве упражнения. Указанный алгоритм, очевидно, работает за линейное время, поэтому и общая сложность решения — $O(n)$. Затраты памяти тоже линейны.

Задача 1-2 (25 баллов). *Правильной скобочной последовательностью* называется строка, состоящая только из скобок, в которой все скобки можно разбить на пары таким образом, что:

- в каждой паре есть левая и правая скобка, причем левая скобка расположена левее правой;
- для любых двух пар скобок либо одна из них полностью внутри другой пары, либо промежутки между скобками в парах не пересекаются
- в паре с круглой скобкой может быть только круглая скобка, с квадратной — квадратная, с фигурной — фигурная

Примеры:

- Если разрешены только круглые скобки:
  - правильные последовательности: `()`, `(())`, `()()`, `()()()`, `(())()`, `((()))`
  - неправильные последовательности: `)(`, `))`, `((`, `())()(`, `()),` `))((`
- Если разрешены круглые и квадратные скобки:
  - правильные последовательности: `[],()`, `[()]`, `[[([])]()]`
  - неправильные последовательности:]), ([)], (())()[]][
- Если разрешены еще и фигурные скобки:

– правильные последовательности: `[{(())}({})]`, `[]{}()`, `{}`, `()`, `[]`

– неправильные последовательности: `[{(})]`, `[(])()]{}`

Во входе задана непустая строка α длины не более 1 000 000, состоящая только из скобок (круглых, квадратных и/или фигурных). Требуется определить, является ли она правильной скобочной последовательностью. Если да, выведите слово `CORRECT`. Если нет, выведите длину максимального префикса α, который либо сам является правильной скобочной последовательностью, либо может быть продолжен до таковой.

Например, для строки `(()))` ответ 4, так как строка `(())` является правильной скобочной последовательностью, а строку `(()))` уже нельзя никаким образом продолжить вправо, чтобы получить правильную скобочную последовательность. Для строки `])()(` ответ 0, поскольку строку `]` нельзя продолжить вправо, чтобы получить правильную скобочную последовательность. Для строки `[(()){()()[[]]}]` ответ `CORRECT`.

Пример входа	Пример выхода
`(())`	`CORRECT`
`([)]`	2
`(([{`	4

Решение.

Рассмотрим первую правую скобку, встречающуюся в строке. Если она находится на первой же позиции, то строка уже некорректна, и нельзя ничего приписать справа, чтобы она стала корректна (не найдется пары к самой первой правой скобке).

Иначе, пусть данная скобка встречается на позиции $r > 0$. Скобка на позиции $r - 1$ является левой, и она является парной к данной правой скобке. Действительно, если бы какая-то левая скобка на позиции $l < r - 1$ была парой к правой скобке на позиции r, то пара к левой скобке на позиции $r - 1$ находилась бы правее r, тогда эти две пары пересекаются, что противоречит определению. С другой стороны, если самая первая правая скобка стоит на позиции $r > 0$, то строку еще можно продолжить так, чтобы она стала корректной: нужно дописать правых скобок столько, чтобы закрыть все левые скобки, и остановиться. Поэтому самая первая правая скобка и левая скобка сразу перед ней соответствуют друг другу. Теперь, если выкинуть эту пару скобок из строки, корректность оставшейся строки определяется тем же самым способом, так как пара скобок, стоящая непосредственно рядом, без промежуточных скобок, не мешает выполнению ни одного из наложенных условий. Из этих соображений вытекает следующий алгоритм решения задачи.

Из определения корректной скобочной последовательности следует, что при чтении такой последовательности слева направо всегда есть набор открывающих скобок, ожидающих себе парных закрывающих скобок (по первому свойству). При этом если встречается очередная закрывающая скобка, она должна быть парной именно к последней открывающей скобке, иначе нарушилось бы второе свойство. В тот момент, когда для последней открывающей скобки в наборе находится парная закрывающая, эта открывающая скобка из набора удаляется, ожидающих открывающих скобок становится на одну меньше, и последней становится та, что была предпоследней. В конце, когда мы прочитали все символы строки, стек должен оказаться пустым: к каждой открывающей скобке нашлась парная. Кроме того, никогда в процессе обработки не должно получиться, что

мы читаем закрывающую скобку, а стек либо пуст, либо на его вершине находится неправильная открывающая скобка. В этом случае последовательность сразу же получается некорректной, и к ней уже справа ничего нельзя дописать, чтобы последовательность стала корректной.

Такое поведение ожидающих открывающих скобок позволяет хранить их в стеке, используя следующий алгоритм. Изначально стек пустой. Идем по строке слева направо, считывая по одному символу. В зависимости от очередного символа, выполняем действия.

- Если очередной символ — открывающая скобка, то кладем его на стек, двигаемся дальше.
- Если очередной символ — закрывающая скобка, то посмотрим на состояние стека. Если на стеке нет ни одного элемента или открывающая скобка на вершине стека не соответствует текущей закрывающей скобке, то последовательность неправильная. При этом она могла быть дополнена до правильной последовательности до текущего момента (обоснуйте), а после того, как найдена неправильная закрывающая скобку, уже не может быть дополнена до правильной. Соответственно, нужно вывести длину прочитанного префикса, не включая последнюю прочитанную скобку, и выйти из программы. Если стек не пуст, и в вершине стека лежит соответствующая открывающая скобка, удаляем из стека эту скобку и двигаемся дальше.

В конце, если пройдена вся строка до конца, нужно еще не забыть определить, является ли прочитанная строка правильной скобочной последовательностью или же только началом правильной скобочной последовательности. Если стек в конце просмотра строки пуст, то прочитанная строка является правильной скобочной последовательностью, и нужно выдать `CORRECT`. Если же стек не пуст, то строку можно дополнить до правильной скобочной последовательности, но она еще такой не является, поэтому нужно выдать длину всей строки.

Для анализа каждого очередного символа строки требуется константное время, поэтому общая сложность алгоритма составляет $O(n)$. Памяти требуется $O(n)$ на хранение самой строки и $O(n)$ на стек.

Задача 1-3 (35 баллов). Вам дано несколько кубиков, каждый задается длинами трех сторон, a, b и c. Считается, что один кубик можно вложить в другой, если их можно так расположить в пространстве, чтобы каждая грань одного кубика была параллельна какой-то грани другого кубика и чтобы при этом один из кубиков полностью содержался внутри другого. Общих точек на границе у них при этом также не должно быть. Нужно определить, какую максимальную цепочку кубиков $C_1, C_2, \ldots, C_k$ можно выбрать из данных таким образом, чтобы C_1 можно было вложить в C_2, C_2 — в C_3 и так далее.

В первой строке входного потока дано число n ($1 \le n \le 1000$). В следующих n строках описаны кубики, на каждой по три целых положительных числа, не превосходящих 10^9, описывающих один кубик. В каждой строке числа выписаны по возрастанию, и первое число следующей строки всегда не меньше первого числа предыдущей. В выходной поток нужно вывести одно число: длину максимальной цепочки вложенных кубиков.

Пример входа	Пример выхода
4 1 1 1 2 2 2 3 3 3 3 3 4	3

Решение.

Заметим, что если кубик с номером i можно вложить в кубик с номером j, то $i < j$. Это следует из того, что стороны кубиков упорядочены, а кубики сами по себе упорядочены по возрастанию наименьшей стороны. Можно доказать, что если есть два кубика, один со сторонами $a_1 \le b_1 \le c_1$, а другой — со сторонами $a_2 \le b_2 \le c_2$, то один из них можно вложить в другой тогда и только тогда, когда $a_1 < a_2, b_1 < b_2, c_1 < c_2$. В частности, если кубики упорядочить по наименьшей стороне, то можно вкладывать только кубик слева в кубик справа.

Это дает нам возможность решать задачу аналогично задаче LIS. Для каждого кубика определим длину $L[i]$ наибольшей цепочки вложенных кубиков, заканчивающейся на этом кубике (т.е. в этот кубик будут вложены все остальные кубики цепочки). Это делается проходом по всем кубикам слева направо и подсчетом $L[i] = \max_{j=0,1,\ldots,i-1;C[j].a<C[i].a,C[j].b<C[i].b,C[j].c<C[i].c} L[j] + 1$, где C - массив с кубиками, a, b и c - наименьшая, средняя и наибольшая стороны кубиков. Алгоритм работает за $O(n^2)$ и требует $O(n)$ памяти.

Задача 1-4 (35 баллов). Пусть задан массив из n целых чисел. По этому массиву будут ходить два указателя l и r ($1 \le l, r \le n$). Изначально оба они указывают на первый элемент массива ($l = r = 1$). Оба указателя могут двигаться только вправо, на одну позицию за раз. При этом указатель l никогда не оказывается правее указателя r, и ни один из них не выходит за пределы массива. Вам нужно после каждого перемещения указателя определить максимум всех элементов от указателя l вправо до указателя r (включая позиции, на которые указывают l и r).

В первой строке входного потока задано число n ($1 \le n \le 100\,000$) — размер массива. Во второй строке n целых чисел от $-1\,000\,000\,000$ до $1\,000\,000\,000$ — сам массив. В третьей строке указано число m ($0 \le m \le 2n - 2$) — количество перемещений. В четвертой строке — m символов `L` или `R`, разделенных пробелами. `L` означает, что нужно сдвинуть l вправо, `R` — что нужно сдвинуть r вправо. Выведите в одну строку ровно m чисел, где i-е число — максимальное значение на отрезке от l до r после выполнения i-й операции.

Указание. Учетная стоимость обработки каждого запроса на перемещение и подсчет максимума должна оказаться $O(1)$.

Пример входа	Пример выхода
10 1 4 2 3 5 8 6 7 9 10 12 R R L R R R L L L R L L	4 4 4 4 5 8 8 8 8 8 8 6

Решение.

Будем хранить индексы, на которые указывают l и r в массиве — так их и назовем для простоты. Будем также хранить дек максимумов текущего окна. А именно, в деке у нас будет находиться невозрастающая последовательность. Самый первый элемент в ней будет совпадать с максимумом всех элементов в текущем окне. Второй элемент, если таковой будет, должен совпадать с максимумом из всех элементов правее самого большого в окне. Т.е. если M — максимальное значение всех элементов окна, то найдем самую левую позицию в окне, на которой стоит число M (их может быть несколько, т.к. в массиве могут быть одинаковые элементы). После этого найдем максимум всех элементов справа от этой позиции в окне — это и будет второй элемент. В частности, он может совпадать с первым, если где-то еще раз встречается число M. Он не может быть больше первого, т.к. иначе максимум всех элементов окна был бы больше, чем M. А может и не быть второго элемента, если M единственный раз встречается на самом правом краю окна. Третий элемент определяется аналогично второму: максимум из всех элементов правее второго элемента, и т.д. Если мы сможем поддерживать такой дек с учетной стоимостью $O(1)$ на операцию сдвига l или r, то мы сможем легко выдать все необходимые по условию максимумы, просто выдавая первый элемент дека после каждой такой операции.

Как нам поддерживать такой дек? Для удобства будем хранить в деке не сами значения элементов последовательности, а их индексы в исходной последовательности. Т.е. первым элементом будет индекс максимального элемента окна, вторым будет индекс максимального из всех справа в окне и т.д. Нам нужно только уметь поддерживать правильный дек при сдвигах l и r.

Сдвиг l делается очень просто. При сдвиге l мы либо сдвигаем левый край окна правее первого элемента дека, либо нет. Если да, то нам нужно удалить первый элемент дека, иначе ничего не нужно трогать. А определить, какой из двух случаев произошел, очень легко: мы знаем индекс, на который указывает l, и знаем индекс, на который указывает первый элемент дека, их нужно просто сравнить. Очевидно, это работает за $O(1)$, даже не учетное, а обычное.

Сдвиг r делается немного сложнее. В окно попадает новый элемент. Он может оказаться больше некоторых из максимумов, на которые указывают элементы в деке. Тогда нужно часть из них выкинуть из конца дека. А именно, нужно выкинуть все элементы в деке, которые указывают на элементы последовательности, строго меньшие, чем новый элемент, добавляемый в окно. Это следует просто из определения нашего дека: новый элемент стал максимальным справа в окне от первого элемента, большего или равного ему, из тех, на которые указывает дек — и все элементы правее в деке больше не нужны. После удаления нужно добавить новый элемент в дек в любом случае. Эта операция уже не обязательно работает за обычное $O(1)$, т.к. нам, быть может, придется много элементов выкинуть из дека. Но она работает за учетную стоимость $O(1)$, т.к. в процессе всего алгоритма мы каждый элемент массива максимум один раз добавляем в дек и максимум один раз удаляем из дека, соответственно на каждый элемент реально уходит максимум две операции. Т.к. задействованных в процессе выполнения m операций сдвига элементов массива $O(m)$, то время на выполнение m операций равно $O(m)$, соответственно учетная стоимость каждой из них — $O(1)$. Затраты памяти этого алгоритма $O(n)$.