Задача 4-1 (60 баллов). Во входе задано бинарное дерево поиска, такое что для любой вершины все ключи в ее левом поддереве строго меньше ее ключа, а все ключи в ее правом поддереве не меньше ее ключа. В первой строке — количество вершин в дереве, во второй — ключи дерева в порядке preorder. В выход выведите две строки. В первой — дерево в порядке postorder, во второй — дерево в порядке inorder. Количество вершин дерева не более 100 000. Значения ключей от 0 до 1 000 000 000.

Пример входа	Пример выхода
7 4 2 1 3 6 5 7	1 3 2 5 7 6 4 1 2 3 4 5 6 7
6 5 3 2 3 5 6	2 3 3 6 5 5 2 3 3 5 5 6

Решение.

Задачу предполагалось решать за $O(n)$, поэтому решение, которое просто строит бинарное дерево поиска из заданных ключей, вставляя ключи по одному, не проходит. Его сложность в худшем случае, когда дерево представляет собой цепочку длины n, равна $O(n^2)$.

Существует довольно много решений, работающих за линейное время, однако все их нужно реализовывать очень аккуратно, чтобы не допустить ошибку. По этой задаче было наибольшее количество неправильных решений и реализаций с багами среди всех задач за семестр.

Задача очевидным образом распадается на две: построение бинарного дерева поиска и вывод его в двух других порядках. После того, как дерево в каком-то виде построено, вывод его в порядках postorder и inorder не представляет проблемы. Вывод в каждом порядке делается с помощью одной рекурсивной функции, которая только вызывает себя для левого и правого сыновей, если они существует, и при этом в нужный момент (после рекурсивных вызовов или между ними) печатает ключ в текущей вершине. Это происходит за линейное время, т.к. для каждой вершины делается $O(1)$ действий.

Интерес представляет первая часть: построение бинарного дерева поиска. Я опишу решение, которое мне кажется наиболее простым, в котором сложнее всего допустить ошибку, что подтверждается решениями студентов, которые брались реализовывать именно это решение. Давайте заведем рекурсивную функцию, которая будет строить поддерево, задающееся данной вершиной. Сигнатура функции будет выглядеть следующим образом:

```
int BuildSubTree(const vector<int>& preorder,
                 int node_index,
                 int range_min,
                 int range_max);
```

В нее передается весь preorder, индекс текущей вершины (т.е. реально номер числа в исходном обходе дерева в порядке preorder, если нумеровать с нуля) и два числа, ограничивающие все ключи в вершинах поддерева снизу и сверху. Зачем эти два числа? Можно понять и доказать, что значения всех ключей в любом поддереве дерева поиска ограничены снизу и сверху. Причем эти границы снизу и сверху однозначно определяются

по значениям ключей всех вершин на пути от корня поддерева до корня всего дерева (кроме самого корня поддерева, оно ничего не ограничивает). Действительно, если мы рассмотрим поддерево с корнем в корне всего дерева, то значения ключей в нем никак не ограничены, могут быть от `INT_MIN` до `INT_MAX`. Если мы рассмотрим левого сына корня всего дерева, то значения ключей в его поддереве уже не могут быть произвольными: они обязаны быть строго меньше ключа в корне. Поэтому в поддереве левого сына корня значения ключей — от `INT_MIN` до `root.value - 1` включительно. В поддереве правого сына корня — соответственно от `root.value` до `INT_MAX` включительно. Если пойти дальше и посмотреть, например, на правого сына левого сына корня, то в нем значения уже ограничены с двух сторон: они должны быть все меньше ключа корня, при этом еще не меньше ключа левого сына корня. Т.е. все значения — от `root.left.value` до `root.value - 1` включительно. А в левом сыне левого сына корня — значения от `INT_MIN` до `root.left.value - 1`. Таким же точно образом, каждый раз, когда мы спускаемся на один уровень вниз, появляется одно дополнительное ограничение на значения ключей. В зависимости от того, идем мы при этом в левого сына или в правого, это ограничение является ограничением сверху или снизу.

Соответственно, т.к. мы должны прочитать дерево из порядка preorder, то все вершины у нас идут в порядке корень — левый сын — правый сын. Запустим нашу процедуру изначально с аргументами `BuildSubTree(0, INT_MIN, INT_MAX)` (вершины дерева мы нумеруем соответственно индексам их ключей в списке preorder). Процедура будет записывать значение ключа очередной вершины, затем будет вызывать себя рекурсивно для построения левого и правого поддеревьев и на выходе из рекурсивных вызовов будет расставлять ссылки из текущей вершины на ее левое и правое поддеревья. Возвращать функция будет количество вершин в поддереве(в частности 0, если первый же ключ уже не лежит в указанном диапазоне от `range_min` до `range_max`). Как ей поможет это возвращаемое значение? См. код и дальнейшее объяснение ниже.

```
int BuildSubTree(const vector<int>& preorder,
                 int node_index,
                 int range_min,
                 int range_max) {
  if (node_index >= preorder.size()) {
    return 0;
  }
  int value = preorder[node_index];
  if (value > range_max || value < range_min) {
    return 0;
  }
  all_nodes[node_index].value = value;
  int left_subtree_size =
      BuildSubTree(node_index + 1,
                   range_min,
                   value - 1);
  if (left_subtree_size != 0) {
    all_nodes[node_index].left = node_index + 1;
  } else {
```

```
    all_nodes[node_index].left = -1;
  }
  int right_subtree_size =
      BuildSubTree(node_index + 1 + left_subtree_size,
      value,
      range_max);
  if (right_subtree_size != 0) {
    all_nodes[node_index].right = node_index + 1 + left_subtree_size;
  } else {
    all_nodes[node_index].right = -1;
  }
  return left_subtree_size + right_subtree_size + 1;
}
```

Все вершины содержат по три поля: ключ, индекс левого сына в общем списке вершин и индекс правого сына в общем списке вершин. Если индекс равен -1, значит соответствующего сына у вершины нет. Общий список вершин заранее создается как `vector` размера n, чтобы не делать динамического выделения памяти, которое и работает долго, и с большой вероятностью приводит к ошибкам выделения/освобождения памяти.

Итак, функция возвращает размер прочитанного поддерева. Вначале мы проверяем, корректен ли текущий индекс и попадает ли соответствующее значение в заданный диапазон. Если что-то из этого не выполняется, значит соответствующее поддерево пусто, и мы возвращаем ноль. Если же поддерево содержит в себе хотя бы корень, то мы знаем значение в корне, знаем, что далее сначала полностью выписано его левое поддерево, а затем правое поддерево. Пытаемся аналогичным образом с помощью рекурсивного вызова прочесть левое поддерево, запоминаем его размер: по нему мы сможем определить, с какого момента в списке начинается запись правого поддерева, т.к. оно должно быть записано сразу же после левого поддерева. Если левое поддерево пустое, устанавливаем указатель на левого сына в никуда, иначе указываем на следующую вершину в списке, ибо именно она обязана быть корнем левого поддерева. Далее читаем с нужного места правое поддерево и проделываем те же махинации. Возвращаем суммарное количество прочитанных вершин в дереве: размер левого поддерева плюс размер правого поддерева плюс корень.

Решение работает корректно, т.к. основывается на определении порядка preorder. Вершина не может попасть не в свое поддерево, т.к. мы указываем полностью весь диапазон возможных значений ключей в поддереве. Если мы зашли в очередной вызов `BuildSubTree` с каким-то диапазоном (`range_min`, `range_max`), то все следующие вершины, попадающие в этот диапазаон, принадлежат текущему поддереву, а как только мы встречаем вершину, не попадающую в этот диапазаон,— поддерево заканчивается.

Решение работает за линейное время, т.к. несмотря на возможность вызова `BuildSubTree` несколько раз с одним и тем же `node_index`, мы просто в процессе обходим дерево в порядке preorder. Кроме этого мы еще иногда вызываем `BuildSubTree` с такими параметрами, для которых поддерево — пустое, однако на каждую вершину дерева есть не более двух “ложных” несуществующих поддеревьев, которые мы проверим за $O(1)$, поэтому мы потратим не более $O(n)$ операций помимо обхода всех вершин в порядке preorder. А обход дерева включает в себя каждую вершину один раз, в каждой

из них тратит время $O(1)$, поэтому все происходит за время $O(n)$. Памяти требуется $O(n)$ для хранения всех трех порядков дерева.

Задача 4-2 (65 баллов). В первой строке входа число n ($1 \le n \le 400$). Во второй строке — n целых чисел $a_1, a_2, \ldots, a_n$. Посчитайте количество различных бинарных деревьев поиска, состоящих из n вершин с ключами $a_1, a_2, \ldots, a_n$. Рассматриваются только такие деревья поиска, у которых в левом поддереве ключи строго меньше, а в правом — не меньше, чем в корне.

Когда два дерева считаются одинаковыми — интуитивно понятно: если нарисовать их на картинке, то картинки должны выглядеть одинаково. Формально это можно определить, например, так: два дерева считаются одинаковыми, если их preorder обходы порождают одну и ту же последовательность чисел. Ответ может быть очень большим числом, поэтому посчитайте его по модулю 123 456 789. Значения ключей от -2^{31} до $2^{31} - 1$.

Пример входа	Пример выхода
2 2 1	2
3 10 10 10	1
3 1 2 3	5

Решение.

Задача решается методом динамического программирования. Хитрость в этой задаче заключается в том, что ключи могут быть одинаковыми. Сначала разберем случай, когда все ключи гарантированно разные, а затем перейдем к более сложному случаю.

Итак, пусть все ключи разные. Пусть их всего n. Очевидно, от того, какие именно n различных ключей у нас имеется, ничего не зависит, можно считать, например, что это — все числа от 1 до n. Тогда устроим динамическое программирование, которое для каждого k от 1 до n найдет нам количество деревьев с k различными ключами. Для $k = 1$ ответ, очевидно, единица. Теперь пусть мы знаем ответы для меньших k, и хотим узнать ответ для текущего k. У нашего дерева обязательно есть корень, и его ключ может совпадать с любым из k ключей. Переберем в цикле, какой из ключей находится в корне. После этого все вершины однозначно делятся на две группы: вершины с ключами, меньшими ключа корня, идут в левое поддерево, остальные — в правое. При этом левое и правое поддеревья уже сами по себе совершенно независимы друг от друга и от корня и могут быть устроены как угодно. Мы знаем размеры левого и правого поддеревьев: если в корне ключ v, то в левом поддереве $v-1$ ключ, а в правом — $k-v$. При этом поддеревья совершенно произвольные, поэтому если мы считаем ответы в массиве ans, то в случае, если в корне находится ключ v, к ответу $ans[k]$ прибавляется $ans[v-1] \cdot ans[k-v]$. Итого, формула для подсчета ответа

$$ans[k] = \sum_{v=1}^{k} ans[v-1] \cdot ans[k-v] \ (mod\ 123456789)$$

Алгоритм работает за $O(n^2)$, т.к. на подсчет ответа $ans[k]$ уходит $O(k)$ времени.

В случае если возможны одинаковые ключи все сложнее: когда мы выбираем корень дерева, в нем может быть не любой ключ. А именно, если, скажем, среди ключей есть три двойки, то только одна из них на самом деле может быть корнем. Можно считать эти двойки различными, в каком-то порядке их занумеровать, чтобы была первая двойка, вторая двойка и третья двойка. При сравнениях будем считать, что вторая двойка больше первой и третья двойка больше второй (т.к. нам все равно в случае равенства какую-то вершину нужно отправить направо, другую — налево). Тогда только первая двойка может быть корнем. Почему? Потому что если двойка в корне, то все остальные двойки обязаны быть в правом ее поддереве. Средняя двойка не может оказаться в корне, так чтобы первая оказалась в левом поддереве. И третья двойка не может оказаться в корне по тем же причинам. Поэтому получается, что количество различных деревьев зависит уже не только от количества ключей, но и от того, какие из них равны, сколько есть групп равных ключей, какого они размера. Соответственно, аналогичная динамика не проходит. Что же делать?

Воспользуемся немного другой динамикой. Отсортируем все ключи. Можно понять, что все ключи в любом поддереве бинарного дерева поиска, построенного из данных ключей, представляют собой непрерывный подотрезок нашего отсортированного массива с ключами. Т.е. все дерево представляет собой весь отрезок ключей — весь массив. Левое поддерево корня образует какой-то префикс всего массива, правое поддерево корня — суффикс всего массива. Поддерево правого сына левого сына корня состоит из всех ключей на отрезке [`root.left.value`, `root.value`], соответственно тоже образует подотрезок всего отсортированного массива ключей, и т.д.

Тогда будем решать следующую задачу: для каждой пары индексов (i, j), $i <= j$ посчитаем, сколько существует деревьев, составленных из всех ключей на отрезке от i-го до $j-1$-го включительно (если $i = j$, то пустое дерево). Ответ к задаче будет храниться в элементе $ans[0][n]$. Как же нам это посчитать? Аналогично первому решению, какой-то ключ — в корне. В отличие от первого случая, не любое значение может быть в корне. А могут быть ровно те, которые являются первыми в списке нескольких подряд идущих одинаковых значений (если какое-то значение повторяется только один раз, то, конечно, это значение может быть в корне, а вот если подряд идет, скажем, три двойки, то только первая из них может оказаться в корне). Пройдем циклом по всем индексам от i до $j-1$ и каждого по очереди назначим корнем (точнее, если $k = i$ или $A[k] \neq A[k-1]$, то $A[k]$ может быть в корне, иначе — не может). Если мы назначили корнем k-го, то все элементы от i-го до $k-1$-го обязаны оказаться в левом поддереве, а все элементы от $k+1$-го до $j-1$-го обязаны оказаться в правом поддереве. Если считать ответ в порядке увеличения длины отрезка от i до j, то к моменту вычисления $ans[i][j]$ мы уже будем знать $ans[i][k]$ и $ans[k+1][j]$, поэтому просто добавим к ответу $ans[i][k] \cdot ans[k+1][j]$. Таким образом мы можем заполнить весь массив:

$$ans[i][j] = \sum_{\substack{i \le k \le j-1 \\ k=i \ or \ A[k] \neq A[k-1]}} ans[i][k] \cdot ans[k+1][j] \ (mod\ 123456789)$$

Изначальные значения $\forall\ 0 \le i \le n-1\ ans[i][i] = 1$ (из 0 ключей единственным способом получается пустое дерево), $ans[i][i+1] = 1$ (дерево из одного элемента тоже всегда одно).

Решение работает за $O(n^3)$, т.к. всего нам нужно заполнить $O(n^2)$ ячеек массива, и каждая заполняется за $O(n)$, используя значения, вычисленные на предыдущих шагах.

Памяти требуется $O(n^2)$ на массив с ответами.

Здесь классический случай перекрытия по подзадачам — один из основных признаков динамического программирования. Оптимальности по подзадачам здесь как таковой нет, т.к. мы ничего и не оптимизируем в этой задаче, но можно к подсчету точного значения относиться как к подсчету, скажем, максимума из одного значения. По сути это та же оптимальность для подзадач, только вырожденная. Если бы мы решали ту же задачу просто рекурсивной функцией, без запоминания и без динамики, то решение работало бы за экспоненциальное время, т.к. одна и та же задача для пары индексов (i, j) решалась бы очень много раз. За счет запоминания ответов мы каждую такую задачу решаем только один раз и добиваемся полиномиальной сложности.

Чтобы не происходили переполнения, нужно хранить значения $ans[i][j]$ в `int64`, а после каждого сложения и умножения сразу брать по модулю 123456789. Тогда при каждом очередном действии оба числа будут меньше 2^{31}, а значит после проведения операции все еще будут влезать в `int64`.

Задача 4-3 (75 баллов).

Реализуйте следующий класс для хранения множества целых чисел:

```
class FixedSet {
 public:
  FixedSet();
  void Initialize(const vector<int>& numbers);
  bool Contains(int number) const;
};
```

`FixedSet` получает при вызове `Initialize` набор целых чисел, который впоследствии и будет хранить. Набор чисел не будет изменяться с течением времени (до следующего вызова `Initialize`). Операция `Contains` возвращает `true`, если число `number` содержится в наборе. Мат. ожидание времени работы `Initialize` должно составлять $O(n)$, где n — количество чисел в `numbers`. Затраты памяти должны быть порядка $O(n)$ в худшем случае. Операция `Contains` должна выполняться за $O(1)$ в худшем случае.

С помощью этого класса решите модельную задачу: во входе будет дано множество различных чисел, а затем множество запросов — целых чисел. Необходимо для каждого запроса определить, лежит ли число из запроса в множестве.

В первой строке входа — число n — размер множества. $1 \le n \le 100\,000$. В следующей строке n различных целых чисел, по модулю не превосходящих 10^9. В следующей строке — число q — количество запросов. $1 \le q \le 1\,000\,000$. В следующей строке q целых чисел, по модулю не превосходящих 10^9. Для каждого запроса нужно вывести в отдельную строку “Yes” (без кавычек), если число из запроса есть в множестве и “No” (без кавычек), если числа из запроса нет в множестве. См. примеры.

Пример входа	Пример выхода
3 1 2 3 4 1 2 3 4	Yes Yes Yes No
3 3 1 2 4 10 1 5 2	No Yes No Yes

Решение.

Нужно реализовать либо вариант `FixedSet` с лекций с помощью двухуровнего хеширования, либо вариант с семинаров с помощью оптимальной совершенной хеш-функции, сохраняющей порядок.

Задача 4-4 (55 баллов). Вам дан набор из n треугольников ($1 \le n \le 1000000$). Каждый треугольник задается тремя целыми числами — длинами его сторон. Длины сторон — положительные целые числа, не превосходящие 10^9. Некоторые пары треугольников подобны. В силу свойств подобия известно, что все эти треугольники распадаются на несколько классов, в каждом из которых все треугольники подобны друг другу. К примеру, три треугольника со сторнами $(6, 6, 10)$, $(15, 25, 15)$, $(35, 21, 21)$ все попарно подобны друг другу, то есть образуют один класс. Необходимо найти количество классов подобия, на которые распадаются данные n треугольников. В первой строке входа дано число n. В каждой из следующих n строк — по три целых числа, задающих стороны очередного треугольника. Выведите число классов подобия. Сложность алгоритма должна в среднем линейно зависеть от n (может еще зависеть от ограничений на стороны треугольника). Т.е. никаких сортировок длинных массивов в решении делать нельзя.

Пример входа	Пример выхода
3 6 6 10 15 25 15 35 21 21	1
4 3 4 5 10 11 12 6 7 8 6 8 10	3

Решение.

Найдем для каждого класса подобных треугольников канонического представителя. Пусть (a, b, c) — какой-то треугольник в классе, будем сразу считать, что $a \le b \le c$. Пусть $g = GCD(a, b, c)$ — наибольший общий делитель чисел a, b и c. Пусть $(a', b', c') = (\frac{a}{g}, \frac{b}{g}, \frac{c}{g})$. Тогда (a', b', c') и будет каноническим представителем класса. Можно доказать, что если взять любой другой треугольник $(a_1, b_1, c_1), a_1 \le b_1 \le c_1$ в том же классе и поделить все

его стороны на их наибольший общий делитель, то получится тот же самый треугольник (a', b', c'). Это оставляется в качестве упражнения читателю.

Теперь будем хэшировать уже канонических представителей. Если у нас будет хэш-функция, которая различные треугольники отображает в различные числа, мы просто раскидаем все треугольники по ячейкам в соответствии с каноническим представителем, и по пути посчитаем количество занятых ячеек. Однако мы не можем на 100 процентов рассчитывать, что у нашей хэш-функции не будет коллизий, мы только можем рассчитывать на хэш-фукнцию, у которой вероятность коллизии достаточно мала. Вспомним пример хэш-функции с семинара, которая применялась для хэширования строк — хэш-функция вида $a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_nx^n (mod\ p)$. В данном случае можно рассматривать канонический треугольник как строку длины 3 — получим хэш-функцию $a_0 + a_1x + a_2x^2 (mod\ p)$. Мы знаем оценку сверху на вероятость коллизии: $\frac{n}{p-1} = \frac{3}{p-1}$. Т.е. видно, что чем больше p, тем меньше вероятность коллизии, т.е. тем лучше.

Сначала опишем, как мы в принципе собираемся решать задачу: заведем хэш-таблицу, которая будет состоять из списков треугольников. Для каждого треугольника считаем канонический треугольник, считаем хэш, добавляем в соответствующую ячейку (для этого нужно взять значение хэш-функции по модулю m, где m — размер таблицы, о нем поговорим ниже). При добавлении проверяем, нет ли уже такого же треугольника в списке. Если есть, то новый элемент не добавляем, если есть — добавляем. На то, чтобы проверить, есть ли в списке наш элемент, требуется O(длины списка), но почти наверняка списки всегда будут маленькой длины, поэтому общая сложность решения (помимо подсчета наибольшего общего делителя) почти наверняка составляет $O(n)$, т.к. на каждый треугольник тратится константное время. Подсчет наибольшего общего делителя двух чисел a и b проводится за $O(\log(\max(a, b))$, поэтому появляется еще логарифмический множитель, однако он уже зависит от размера сторон треугольников, а не от количества треугольников.

Теперь немного о реализации этого решения. Во-первых, здесь необходимо понимать, что три параметра задачи — n, m и p — это вообще говоря совершенно разные числа, никакие два из них не обязаны совпадать. Числом n мы не управляем, оно приходит извне, мы только знаем ограничение сверху на это число. Числа m и p мы выбираем сами. Число p мы выбираем просто побольше, чтобы вероятность коллизии была поменьше. В частности, можно выбрать $2^{31} - 1$ — максимально возможное значение `int`'а, т.к. это простое число. Число m выбирается из соображений trade-off между количеством имеющейся памяти и скоростью работы. Чем больше m, тем меньше будет заполненность таблицы, а это тоже влияет на вероятность коллизии. Но тем больше и памяти нам потребуется. В данной задаче все это не так уж важно, т.к. она учебная, и не используется как подпрограмма в какой-то большей системе: можно взять любое большое m, которое еще влезает в ограничения по памяти. В каждом конкретном случае этот вопрос приходится решать отдельно.

Еще одно важное замечание относится к подсчету хэш-функции. Если p большое, то при вычислении выражения a_2x^2 переполнится `int`, если считать в лоб. Даже при подсчете a_1x он может переполниться. Чтобы этого избежать, нужно

1. Пользоваться типом `int64 (long long)` для подсчетов
2. После каждого перемножения двух чисел сразу брать результат по модулю p, до

следующего перемножения

3. Сразу складывать по модулю

В частности, выражение a_2x^2 придется вычислить в два этапа, делая сначала одно умножение, беря по модулю и делая еще одно умножение.

Идея в том, что в каждый момент времени аргументы для сложения/умножения влезают в `int`, т.к. это остатки по модулю p, которое влезает в `int`, значит после сложения/умножения результат будет влезать в `int64`, а если затем сразу же взять по модулю p, то получится опять число, которое влезает в `int`.