Двоичный (бинарный) поиск

Рассмотрим задачу поиска элемента в массиве. Используя известные нам методы, решение этой задачи по времени сравнимо с O(N). Но если исходный массив уже отсортирован, то элемент в нем можно найти гораздо быстрее, воспользовавшись **двоичным** (*бинарным*) **поиском**. Его идея заключается в делении списка пополам. В зависимости от значения среднего элемента, мы переходим либо к левой, либо к правой половине списка. Тем самым, длина части, в которой мы ищем элемент, сокращается в два раза на каждом шаге, а, значит, общая сложность алгоритма будет $\mathbf{O(\log_2 N)}$.

Итак, перед нами стоит задача — выяснить, содержится ли элемент **key** в некотором списке, или в его части. Мы будем сокращать часть списка, в которой мы ищем элемент key. А именно, введем две границы — *left* и *right*. При этом мы будем знать, что элемент A[right] строго больше, чем key (то же самое можно сказать и про элементы, которые правее right). Про элемент A[left] и те, которые находятся левее него мы будем знать, что все они меньше или равны key. А вот про элементы, которые лежат строго между A[left] и A[right] (то есть про элементы, чьи индексы больше left, но меньше right), мы ничего не знаем.

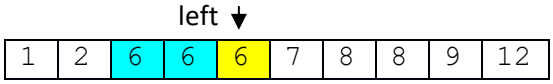
В самом начале мы ничего не знаем про все элементы массива, поэтому присвоим left=-1 и right = len(A). Можно представить это так — к концам массива добавляются два фиктивных элемента, в левый конец добавляется элемент, в который записывается минус бесконечность (т. е. значение, заведомо меньшее, чем key), и этот элемент имеет индекс -1, а в правый элемент дописывается элемент, равный плюс бесконечности, и его индекс равен len(A). Соответственно, переменные left и right первоначально указывают на эти фиктивные элементы (то есть на самом деле никаких элементов к массиву добавлять не требуется, мы это делаем лишь мысленно).
Затем разделим отрезок от left до right на две части и возьмем средний элемент между ними. Его индекс равен middle = (left + right) // 2.

Сравним значение этого элемента со значением key. Если A[middle] строго больше чем key это означает, что сам элемент A[middle] и все, что правее него, должно попасть в правую часть. Это означает, что нужно сделать присваивание right = middle. Иначе (если A[middle] <= key), то элемент A[middle] и все, что левее него, должно попасть в левую часть, то есть необходимо присвоить left = middle.
Будем повторять этот процесс, пока между двумя границами left и right еще есть элементы, то есть пока right > left + 1. Получаем следующий алгоритм:

```
left = -1
right = len(A)
while right - left > 1:
    middle = (left + right) // 2
    if A[middle] > key:
        right = middle
    else:
        left = middle
```

Что будет после завершения этого алгоритма? left и right указывают на два соседних элемента, при этом A[right] > key, A[left] <= key. Таким образом, если элемент key содержится в списке, то A[left] == key. Правда, возможна ситуация, когда left == -1 (если в списке A все элементы строго больше key), поэтому для ответа на вопрос «содержится ли в списке A элемент key» необходимо проверить условие:
left >= 0 and A[left] == key

Т.е. для данного списка A и key = 6 получим left = 4, right = 5, A[4] == key



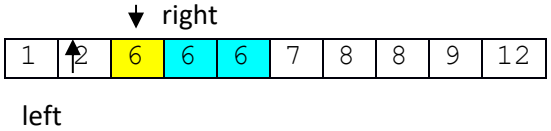
Таким образом, мы нашли не просто какой-то элемент, равный данному, а самый правый. Поэтому еще одно название именно этой реализации алгоритма **«Правый бинпоиск»** и обозначение для запоминания **[ )**

Что же можно сказать про значение right? Это минимальный элемент списка, который строго больше, чем key. Иными словами, на место элемента A[right] можно вставить элемент со значением key (сдвинув при этом всю правую часть списка на один элемент), сохраняя упорядоченность списка, при этом right – самая правая такая позиция. В этом случае говорят, что значение ***right*** является **«верхней границей»** (UpperBound) для элемента key: правее этой позиции нельзя вставить элемент key, сохраняя список упорядоченным.

Теперь модифицируем алгоритм двоичного поиска так, чтобы элементы, в точности равные key, оказывались в правой части списка, а не в левой, как раньше. Это потребует изменения неравенства в условии внутри цикла:

```
left = -1
right = len(A)
while right - left > 1:
    middle = (left + right) // 2
    if A[middle] >= key:
        right = middle
    else:
        left = middle
```

На каких значениях left и right закончится данный цикл? A[left] < key, A[right] >= key.
Т.е. для данного списка A и key = 6 получим left = 1, right = 2, A[2] == key



Таким образом, мы нашли не просто какой-то элемент, равный данному, а именно первый слева. Поэтому еще одно название именно этой реализации алгоритма **«Левый бинпоиск»** и обозначение для запоминания **( ]**

Элемент равный key можно вставить на место A[right], сохраняя упорядоченность списка, и это минимальная позиция, куда его можно вставить. Иными словами, данная функция возвращает **«нижнюю границу»** (LowerBound) для элемента key.
При помощи LowerBound также можно проверить вхождение элемента key в список A при помощи условия `right < len(A) and A[right] == key`

UpperBound возвращает индекс первого элемента, который больше чем key, а LowerBound возвращает индекс первого элемента, который равен key, то значение разности (UpperBound – LowerBound) равно числу вхождений элемента key в список A.
Если элемент key в массиве отсутствует, то и UpperBound и LowerBound будут указывать на одно и то же число.

Для работы на отсортированных данных в Python есть встроенный модуль **`bisect`**. Модуль bisect - обеспечивает поддержку списка в отсортированном порядке с помощью алгоритма деления пополам.

Набор функций:

bisect.insort(list, elem), он же bisect.insort_right(list, elem) - вставка элемента в отсортированный список, при этом elem располагается как можно правее (все элементы, равные ему, остаются слева).

bisect.insort_left(list, elem) - вставка элемента в отсортированный список, при этом elem располагается как можно левее (все элементы, равные ему, остаются справа).

bisect.bisect(list, elem), он же bisect.bisect_right(list, elem) - поиск места для вставки элемента в отсортированный список, таким образом, чтобы elem располагался как можно правее.

bisect.bisect_left(list, elem) - поиск места для вставки элемента в отсортированный список, таким образом, чтобы elem располагался как можно левее.

Для полного счастья не хватает только функции для проверки наличия элемента в отсортированном списке. Можно написать свою функцию проверки

```
from bisect import *

def chek(a, elem):
    index = bisect_left(a, elem)
    if index < len(a):
        return a[index] == elem
    return False
```

Очень часто в решении олимпиадных задачах используют бинарный поиск для нахождения ответа – «бинарный поиск по ответу». Опишем пример, для которого подойдет такой прием. У нас есть некая функция f(x), которая является монотонно неубывающей или невозрастающей для всех x из (A, B). Нам нужно найти такое значение x для которого f(x) = C (как вариант, первое большее C, первое меньшее C, удаленное от C на расстояние не больше eps). В этом случае мы можем представить значения x как индексы некоторого отсортированного списка, значения которого f(x). Далее будем использовать идею бинарного поиска в массиве, описанную раньше. Разница только в том, что мы не создаем такой массив полностью, вычисляя для каждого x значение f(x), а вычисляем только те значения, которые необходимы в процессе решения задачи, а таких значений будет не больше чем $\log_2 (B - A)$.

Примеры решения задач с использованием идеи бинарного поиска

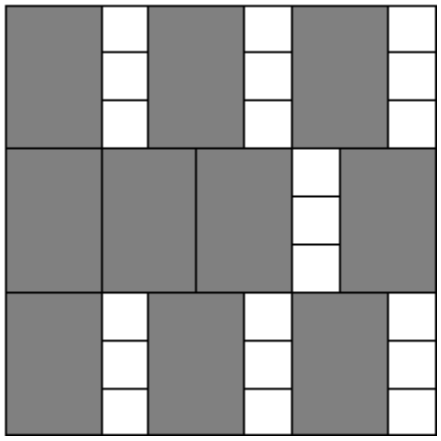
Задача №1. Дипломы

Когда Петя учился в школе, он часто участвовал в олимпиадах по информатике, математике и физике. Так как он был достаточно способным мальчиком и усердно учился, то на многих из этих олимпиад он получал дипломы. К окончанию школы у него накопилось n дипломов, причём, как оказалось, все они имели одинаковые размеры: w — в ширину и h — в высоту.

Сейчас Петя учится в одном из лучших российских университетов и живёт в общежитии со своими одногруппниками. Он решил украсить свою комнату, повесив на одну из стен свои дипломы за школьные олимпиады. Так как к бетонной стене прикрепить дипломы достаточно трудно, то он решил купить специальную доску из пробкового дерева, чтобы прикрепить её к стене, а к ней — дипломы. Для того чтобы эта конструкция выглядела более красиво, Петя хочет, чтобы доска была квадратной и занимала как можно меньше места на стене. Каждый диплом должен быть размещён строго в прямоугольнике размером w на h. Дипломы запрещается поворачивать на 90 градусов. Прямоугольники, соответствующие различным дипломам, не должны иметь общих внутренних точек.
Требуется написать программу, которая вычислит минимальный размер стороны доски, которая потребуется Пете для размещения всех своих дипломов.
Входные данные: три целых числа: w, h, n ($1 \le w, h, n \le 10^9$).
Выходные данные: одно число – ответ задачи.
Пример

Вход	*Выход*
2 3 10	9

Иллюстрация к примеру



Рекомендации по решению

Из-за больших ограничений на n, w, h линейный перебор возможной длины стороны квадрата **ans** не проходит по времени. Обратим внимание на тот факт, что чем больше сторона квадрата, тем больше дипломов на ней можно уместить. Т.е. при увеличении **ans** будет увеличиваться и f(ans) – количество размещаемых дипломов. Поэтому для решения данной задачи можно использовать бинарный поиск по ответу: с помощью бинарного поиска будем подбирать наименьшее значение **ans,** удовлетворяющее условию задачи. За O(1) легко проверить, поместятся ли n дипломов в квадрат со стороной **ans** $n \le$ (ans /w) * (ans / h).

Будем брать среднее значение из возможных длин квадрата ans = (L + R) // 2.
Если такое значение ans позволяет разместить n дипломов, то будем двигать правую границу в ans, т.к. все значения правее R тоже позволят разместить n дипломов, но размеры доски будут больше, чем текущее значение ans. Иначе будем двигать левую границу. Будем продолжать поиск лучшего решения до тех пор, пока L и R не станут соседними числами.
Где же хранится ответ задачи?
Проанализируем наши действия. Во время работы алгоритма решения задачи если значение ans нам подходило, то мы ставили туда R, иначе L. То есть ответ гарантировано лежит в правой границе – R, а в L его быть не может. Для того алгоритм сработал правильно нужно чтобы и начальные значения L и R так же удовлетворяли этому условию: в L такое значение, которое не может быть ответом, а в R – значение гарантирующее размещение всех дипломов. Поддержка начальных условий во время работы алгоритма называется инвариант. Для нашего случая возьмем лучший случай для минимального значения и вычтем единицу: L = min(w, h) –1, и худший случай для максимального: R = n * max(w, h).
Сложность такого решения: $O(\log_2(n * \max(w, h)))$.

Задача №2. Деление пополам

Дано действительное число **a** и натуральное **n**. Вычислите корень n-й степени из числа a. Для решения используйте метод деления отрезка пополам.
Входные данные: число a – действительное, неотрицательное, не превосходит 1000, задано с точностью до 6 знаков после запятой, число n – натуральное, не превосходящее 10.
Выходные данные: единственное число ответ на задачу с точностью не менее 6 знаков после запятой.
Пример

Вход	*Выход*
2 2	1.4142135623730951
81 4	3.0

Рекомендации по решению
Эта задача представляет собой подвид задач на тему бинарный поиск по ответу: мы будем искать корень уравнения $ans = \sqrt[n]{a}$. Так как функция извлечения натурального корня из неотрицательного числа является возрастающей, то для всех xi из области определения функции будет выполняться условие если, x1 < x2, то f(x1) < f(x2). При решении этой задачи будем считать, что у нас нет встроенной функции, позволяющей извлекать корень n-ой степени из числа. Поэтому «перевернем» эту задачу – будем искать такое число ans, такое что $ans^n = a$. Если для текущего ans значение $ans^n > a$, то будем двигать правую границу в ans, иначе левую.
Особенностью подбора ответа для этой задачи – точность. Мы можем не найти такого решения, которое даст нам строгое равенство $ans = \sqrt[n]{a}$, а будет соответствовать ответу только с определенной точностью.
Один вариантов описания цикла для решения этой задачи будет сравнение расстояния между левой и правой границей поиска до тех пор, пока оно не станет меньше EPS. Значение EPS должно быть меньше чем точность, которую нужно получить. Т.е. если необходимо получить точность не менее 6 знаков после запятой, то EPS ≤ 0.0000001. Десятичную дробь можно записать в формате 1e-7, что гарантирует нужное количество нулей после запятой.

Еще один вариант цикла – цикл с параметром. Количество повторений k можно вычислить, зная диапазон возможных значений ответа и учитывая, что при каждой итерации количество вариантов уменьшается в 2 раза, т.е. $k = \log_2 M$, где M – количество возможных значений. Для нашей задачи количество вариантов ответов не менее 10^9. Это значит, что количество итераций должно быть чуть больше 30.

При работе с дробными числами в python нужно помнить, что гарантированная точность вычислений в стандартной комплектации недостаточная для решения многих задач. Убедиться в этом можно на простом примере: 0.3 - 0.2 = 0.09999999999999998.
Если вы подозреваете, что решение вашей задачи на python «падает» на точности, то используйте **decimal** https://docs.python.org/3/library/decimal.html

Используемые ресурсы:

1. Python v3.0.1 documentation. Документация по Python 3.
Сайт: https://docs.python.org/3.0
2. Видеозаписи лекций ЛКШ. Бинарный поиск по ответу. Лектор: Пётр Калинин
Сайт: http://sis.khashaev.ru/2013/july/b-prime/4XnfyOJEnSM/
3. Cbook. Автор Д.П. Кириенко
4. Учебник Foxford по информатике. Лекция «Бинарный поиск в массиве на Python (два фиктивных элемента)» Лектор: В.М. Гуровиц
Сайт: https://foxford.ru/wiki/informatika/binarnyy-poisk-v-massive-na-python-dva-fiktivnyh-elementa