Бинарный поиск в списках

Вам уже знаком алгоритм бинарного поиска и его реализация с помощью цикла `while`. Сегодня мы научимся применять специальный модуль Python, предназначенный для бинарного поиска в отсортированном списке.

1 Модуль bisect

Модуль `bisect` позволяет использовать несколько встроенных функций для бинарного поиска. Для их использования нужен отсортированный по возрастанию список. Точнее по неубыванию, так как некоторые элементы могут совпадать.

1.1 bisect_left

Функция `bisect_left` принимает на вход список и элемент `x`, который мы ищем в списке, и возвращает первую позицию, в которую можно поставить `x`, чтобы сохранить упорядоченность списка.

Рассмотрим несколько примеров:

```
from bisect import bisect_left

arr = [-3, 0, 2, 2, 2, 3, 8, 100]

x = 3
pos = bisect_left(arr, x)
print(pos)
```

```
5
```

Для `x = 3` ответ `5` – позиция, в которой сейчас стоит `3`. Действительно можно подвинуть все элементы вправо, а в эту позицию поставить элемент `3` и упорядоченность сохранится.

```
x = 2
pos = bisect_left(arr, x)
print(pos)
```

```
2
```

Для `x = 2` ответ `2` – первая позиция в которую можно поставить двойку. Делаем вывод: если искомый элемент есть в списке, то `bisect_left` вернет позицию первого элемента, равного данному.

```
x = -100
pos = bisect_left(arr, x)
print(pos)
```

```
0
```

Для `x = -100` ответ `0`. Значит, если элемент меньше минимального элемента списка, то ответ – `0`.

```
x = 101
pos = bisect_left(arr, x)
print(pos)
```

```
8
```

Для `x = 101` ответ `8` – это позиция за последним элементом списка. Действительно, если туда поставить элемент больше максимального, то упорядоченность сохранится и из всех позиций это будет самая левая.

Как поведет себя `bisect_left`, если элемента с переданным в качестве аргумента значением не существует? Поищем позицию вставки `x = 5`:

```
x = 5
pos = bisect_left(arr, x)
print(pos)
```

```
6
```

Ответ – `6`. В этом случае слева останется элемент меньший заданного (`3`), а в эту позицию можно поставить искомый.

Функцию `bisect_left` можно использовать для проверки наличия элемента в отсор-

тированном списке. Она сработает быстрее, чем линейный поиск. Однако если таких проверок нужно делать много, то можно сохранить элементы в словаре или множестве и выполнять проверку эффективнее, чем за двоичный логарифм от количества элементов.

1.2 bisect_right

Функция `bisect_right` принимает на вход список и элемент `x`, который мы ищем, и возвращает последнюю позицию, в которую можно поставить `x` чтобы сохранить упорядоченность списка. Другими словами, мы ищем самую правую позицию, куда можно поставить искомый элемент.

Сравним на одних и тех же примерах работу левого и правого бинпоиска и сделаем несколько выводов:

```
from bisect import bisect_right

arr = [-3, 0, 2, 2, 2, 3, 8, 100]

x = 3
print(bisect_left(arr, x), bisect_right(arr, x))

x = 2
print(bisect_left(arr, x), bisect_right(arr, x))

x = -100
print(bisect_left(arr, x), bisect_right(arr, x))

x = 101
print(bisect_left(arr, x), bisect_right(arr, x))

x = 5
print(bisect_left(arr, x), bisect_right(arr, x))
```

```
5 6
2 5
0 0
8 8
6 6
```

- `bisect_right` никогда не укажет позицию, в которой стоит такой же элемент, потому что всегда можно поставить искомый элемент правее этого.
- если элемента нет в списке, то левый и правый бинпоиск вернут одно и тоже значение для позиции вставки.
- если взять разницу между правым и левым бинпоиском, то получим количество элементов, равных заданному. Это самый эффективный способ посчитать коли-

чество одинаковых элементов: выполняются всего две операции за двоичный логарифм от длины списка, и время работы не зависит от количества элементов.

Можно ли запускать алгоритмы встроенного биноиска на неотсортированных списках? Формально такого ограничения нет. Функции модуля `bisect` не проверяют список на отсортированность и не вызывают ошибки, однако полученный таким образом результат не претендует на правильный ответ.

insort_left и insort_right

Кроме функций поиска левой и правой позиции куда можно поставить элемент, есть и функции вставки.

`insort_left(arr, x)` сначала выполняет функцию `bisect_left()`, чтобы найти точку вставки, а затем запускает метод `insert()`, чтобы вставить `x` в соответствующую позицию для сохранения порядка сортировки.

Функция `insort_right(arr, x)` аналогична функции `insort_left()`, но вставляет `x` в `arr` после всех существующих записей `x`. Эта функция сначала выполняет `bisect_right()`, чтобы найти точку вставки, а затем она запускает метод `insert()`, чтобы вставить `x` в соответствующую позицию для сохранения порядка сортировки.

```
from bisect import insort_right, insort_left

arr = [-3, 0, 2, 2, 3, 8, 100]
insort_left(arr, 4)
print(arr)

arr = [-3, 0, 2, 2, 3, 8, 100]
insort_right(arr, 8)
print(arr)
```

```
[-3, 0, 2, 2, 3, 4, 8, 100]
[-3, 0, 2, 2, 3, 4, 8, 8, 100]
```

Обратите внимание, что время работы этих функций зависит от времени вставки в список. Алгоритм работает так же медленно, как `insert` для неотсортированных списков.

2 Встроенный бинпоиск в олимпиадных задачах

В олимпиадном программировании встроенный бинпоиск менее популярен, чем двоичный поиск по ответу. Но есть ряд задач, в которых данные изначально упорядочены, или их можно представить в виде отсортированного списка. Тогда открывается возможность для применения встроенного бинпоиска.

Посмотрим, как это сделать на примере задачи «k-ый ноль»

Задача

Для последовательности из чисел 0 или 1 даются запросы вывести позицию k-ого по счету нуля. В ответ требуется вывести индекс этого элемента в исходном массиве или -1, если ответа нет.

В первой строке входных данных вводится длина последовательности. Во второй строке вводится сама послежовательность. В третьей строке вводится количество запросов. Затем, в каждой следующей строке вводятся сами запросы. Требуется для каждого запроса вывести ответ.

Входные данные	*Выходные данные*
9 0 1 0 1 1 0 0 1 1 3 4 5 1	6 -1 0

Рассмотрим тестовый пример к задаче. У нас есть последовательность из 9 чисел. К ней 3 запроса:

- 1-й запрос: выведите позицию четвертого нуля в последовательности. Ответ – 6, так как четвертый по счету ноль стоит на 6 позиции
- 2-й запрос: 5 и ответ -1, т.к. в последовательности всего 4 нуля и вывести позицию пятого нуля невозможно
- 3-й запрос: 1, ответ 0. Индекс первого нуля – ноль

Для того, чтобы применить к решению этой задачи бинарный поиск по списку нужно создать какой-то список с неубывающими элементами. Давайте создадим список, в котором сохраним количество нулей последовательности от начала до текущего элемента включительно:

```
count[0] = 1 if a[0] == 0 else 0
for i in range(1, n):
    if(a[i] == 0):
        count[i] = count[i-1] + 1
    else:
        count[i] = count[i-1]
```

Для тестового примера список будет таким:

индекс	0	1	2	3	4	5	6	7	8
`a`	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
`count`	**1**	**1**	**2**	**2**	**2**	**3**	**4**	**4**	**5**

Список `count` неубывающий, что дает нам право использовать встроенный бинпоиск.

Как понять в какой позиции стоит k-ый ноль> Посмотрим на примере поиска второго нуля. В доп. списке левее позиции второго нуля стоит число 1, что означает, что количество нулей до этого было 1. Начиная с этой позиции, оно равно 2. Значит, ровно в той позиции списка `count`, где первый раз появилось число k, и находится k-ый ноль.

Это соответствует работе левого двочиного поиска, который находит первую позицию с элементом, равным нашему. Так как числа доп. списка возрастают на 1, то единственная ситуация, когда поиск не найдет ответ – это если число k больше количества нулей в последовательности.

```
nom = bisect_left(count, k)
if nom == n:
  print(-1)
else:
  print(nom)
```

Вот такой красивый способ решения этой задачи можно придумать, используя встроенный бинпоиск.

3 Сортировка элементов при двоичном поиске

Во всех примерах мы рассматривали списки чисел, отсортированные по значению элементов. Однако, как мы видели в одной из более ранних лекций, сортировка по возрастанию, встроенная в Python, может сортировать и по собственному правилу – ключу сортировки. Список будет отсортирован по возрастанию ключей этих элементов. Встроенные функции бинпоиска также позволяют выполнять бинарный поиск с использованием ключа.

Пусть у нас есть функция `f(x)`, которая сравнивает не сами элементы, а противоположные им по знаку. Такое сравнение позволяет отсортировать элементы по убыванию. Отсортируем исходный список с ключом `f`:

```
def f(x):
    return -x
arr=[5, 7, -1, -9, 45, 2, 79, 2]
arr.sort(key=f)
print(arr)
```

```
[79, 45, 7, 5, 2, 2, -1, -9]
```

Теперь мы также можем выполнить левый и правый бинпоиск. В функцию поиска мы передаем в качестве аргумента не сам элемент, а функцию от этого элемента. Также нам будет нужно указать ключ поиска `key=f`:

```
x = 2
pos_l = bisect_left(arr, f(x), key=f)
pos_r = bisect_right(arr, f(x), key=f)
print(pos_l, pos_r)
```

```
4 6
```

Теперь бинпоиск в отсортированном по правилу `f` списке будет искать левую и правую позицию для постановки элемента `f(x)`. Полученный ответ для `x = 2` совпадает с ожидаемым: 4 позиция соответствуют индексу первой двойки в списке, а 6 позиция – индекс первого элемента большего двойки.

Все рассмотренные сегодня задачи можно решить, написав бинпоиск самому. Но помните, что встроенные функции в Python работают быстрее авторских.