Применения двоичного поиска по ответу

1 Двоичный поиск по ответу с целочисленными значениями

Сегодня мы разберем еще несколько примеров задач на двоичный поиск по ответу. Рассмотрим следующую задачу.

Задача

В вашем распоряжении есть n веревочек, каждая имеет некоторую длину a_i. Вы можете резать веревочки, но не можете их связывать или склеивать. Нужно получить из них x веревочек одинаковой целочисленной длины, причем наибольшей возможной длины.

Заметим, что если мы можем сделать нужное число веревочек длины k, то мы можем сделать не меньше веревочек длины $k-1$, просто отрезав от каждой веревочки единицу длины. Это дает нам возможность воспользоваться двоичным поиском по длине веревочки. Пусть левая граница указывает на подходящее значение длины веревочек, то есть всегда можно набрать хотя бы `x` веревочек длины `left`. Правая граница, соответственно, указывает на неподходящее значение длины. Разберемся с начальными значениями. Мы всегда можем получить бесконечное количество веревочек длины `0` – это наибольшее значение, которое мы гарантированно можем получить. Положим `left = 0`. В качестве правой границы можно выбрать максимальное значение в списке `a` плюс один: очевидно, что у нас нет ни одной такой веревочки.

Функции `check` теперь потребуется весь список исходных веревочек, чтобы проверить очередное значение `k` на допустимость. Переберем все веревочки и прибавим к счетчику, на сколько веревочек длины `k` можно нарезать очередную. Если в итоге получится число не меньшее `x`, значит, такое `k` нам подходит:

```
def check(k, x, n, a):
    s = 0
    for i in range(n):
        s += a[i] // k # столько веревочек длины k можно получить из i-й начальной
    return s >= x
```

Итоговый ответ будет лежать в левой границе двоичного поиска, ведь там подходящее значение:

```
left = 0
right = max(a) + 1
while right - left > 1:
    mid = (left + right) // 2
    if check(mid, x, n, a):
        left = mid
    else:
        right = mid
print(left)
```

Перейдем к следующей задаче.

Задача

Пусть нам даны k коров и n стойл (стойл не меньше, чем коров). У каждого стойла есть своя координата, стойла перечислены в порядке возрастания координат.

Если коровы стоят слишком близко друг к другу, то они начинают веселиться, а это плохо влияет на молоко. Поэтому работники фермы хотят расставить коров по стойлам так, чтобы минимальное расстояние между коровами было как можно больше.

Например, если у нас есть 6 стойл, в позициях 2, 5, 7, 11, 15 и 20, и нам нужно расставить трёх коров, мы можем их расставить в позициях 2, 7 и 15 – тогда минимальное расстояние будет 5 (расстояние между 2 и 7). А если мы расставим коров в позициях 2, 11 и 20, то минимальное расстояние будет 9 (между 2 и 11 и между 11 и 20).

Из условия следует, что нужно расставить коров на максимально возможном расстоянии друг от друга. При этом можно заметить, что если мы не можем расставить коров так, чтобы расстояние между ними было хотя бы x, то мы не сможем расставить их и так, чтобы между ними было расстояние больше x. Аналогичные рассуждения можно произвести в обратную сторону: если мы можем расставить коров, чтобы между ними было расстояние больше либо равное y, то можем расставить и так, чтобы расстояние было больше либо равно $y - 1$ (как минимум мы можем оставить ту же расстановку).

Это снова приводит нас к двоичному поиску по ответу. Давайте в левой границе у нас будет точно возможное расстояние, а в правой — точно невозможное. Стартовое значение для левой границы равно единице (просто поставим коров в k последовательных стойл, между ними гарантировано расстояние не меньше единицы). Возможно, расстояние даже уже больше, но это неважно, двоичный поиск это определит. Правую границу можно задать значением расстояния между крайними стойлами, увеличенного на один. Если коров хотя бы две, то такого расстояния никак не достичь, нет соответствующей пары стойл.

Теперь запустим наш двоичный поиск по ответу:

```
left = 1
right = a[-1] - a[0] + 1
while right - left >= 1:
    mid = (left + right) // 2
    if check(mid, a, k):
        left = mid
    else:
        right = mid
print(left)
```

Осталось научиться проверять, допустимо ли некоторое расстояние x. Давайте будем жадно расставлять коров слева направо. Очевидно, ничем не плохо поместить первую корову в самое левое стойло: если бы существовало подходящее расположение коров, где самая левая корова не в левом стойле, то её можно было бы подвинуть влево, и ответ все еще остался бы оптимальным (расстояние между первой и второй коровами увеличилось, а остальные не изменились).

Поместив первую корову в самое левое стойло, по тем же соображениям поместим вторую корову в самое левое доступное стойло. Стойло считается доступным, если оно находится на расстоянии большем либо равном x от последнего занятого стойла. Будем в цикле поддерживать координату последнего занятого стойла и количество расставленных коров. Если расстояние от текущего стойла до последнего занятого больше либо равно x, поставим очередную корову. Если в итоге мы таким образом можем поставить хотя бы k коров, то такое x будет допустимым – вернем из функции `check` значение `True`:

```
def check(x, a, k):
    res = 1
    last = a[0]
    for i in range(1, len(a)):
        if a[i] - last >= x:
            res += 1
            last = a[i]
    return res >= k
```

Таким образом мы найдем максимальное возможное расстояние, оно будет храниться в левой границе двоичного поиска. Если нам нужно еще и восстановить расположение коров, то запустим аналогичный цикл по коровам, как мы запускали в функции `check`. Только нам нужно остановиться, как только мы расставили `k` коров, и вывести ответ:

```
res = 1
last = a[0]
print(a[0])
for i in range(1, len(a)):
    if a[i] - last >= x:
        res += 1
        last = a[i]
        print(a[i])
        if(res == k):
            break
```

2 Двоичный поиск по ответу с вещественными значениями

Давайте теперь посмотрим, как применять двоичный поиск по ответу в случае, когда ответ может быть вещественным числом. Рассмотрим следующую задачу.

Задача

Пусть вам дают кубическое уравнение, то есть уравнение вида $ax^3+bx^2+cx+d=0$. Требуется найти его корень. То есть, необходимо найти такое x по данным a, b, c, d, что многочлен равен нулю. При этом входными данными гарантируется, что такой x единственный. Например, у уравнения $3x^3+4x^2+2x=0$ единственным корнем является $x=0$.

Раз корень единственный, значит, с одной стороны от корня все значения многочлена положительные, а с другой стороны от корня — отрицательные. Давайте поймем, с какой стороны положительные, а с какой отрицательные. Для этого можно посмотреть знак многочлена в точке, где икс равен плюс бесконечности. Это точка, бесконечно удаленная вправо. В случае бесконечно большого x на знак выражения влияет только знак ax^3, все остальные слагаемые маленькие на фоне бесконечности в кубе. x^3 – положительный, значит, если a положительно, то и весь многочлен справа положительный, и наоборот, если a отрицательно, весь многочлен справа отрицательный.

Мы определили знак многочлена справа. Давайте теперь еще сделаем так, чтобы многочлен справа от корня был всегда положительный, а слева от корня отрицательный. Для этого, в случае если $a<0$, домножим обе части уравнения на -1. Ноль, умноженный на минус один, останется нулем. Слева у каждого слагаемого знак изменится на противоположный – получим уравнение $-ax^3-bx^2-cx-d=0$, и коэффициент при x^3 станет положительным. Корень от этого не изменится, потому что в корне значение равно нулю:

```
if(a < 0):
    a, b, c, d = -a, -b, -c, -d
```

Своими преобразованиями и размышлениями мы свели задачу к тому, что у нас есть

функция, которая сначала отрицательна до некоторого момента, а затем положительна. Как мы уже с вами говорили на занятии по вещественным числам, их не стоит сравнивать на равенство, поэтому будем считать, что все значения функции либо отрицательны, либо положительны. Давайте левая граница двоичного поиска у нас всегда будет указывать на отрицательное число, а правая граница — на положительное. Тогда, если эти две границы будут находиться очень близко друг к другу, можно будет считать, что с некоторой точностью корень уравнения находится между ними.

Как нам теперь найти два значения, между которыми находится корень? Можно попробовать оценить с точки зрения математики, в каких границах может находиться значение корня. А можно положить левую границу равной минус единице, а правую границу равной единице. Если значения функции в этих точках имеют одинаковый знак, то давайте домножим на два оба значения, и таким образом удвоим длину отрезка. Будем делать так, пока не окажется, что значения функции в этих точках имеют разный знак:

```
def f(x):
    return a*(x**3) + b*(x**2) + c*x + d

left = -1
right = 1
while f(left)*f(right) >= 0:
    left *= 2
    right *= 2
```

Теперь, мы убедились что в точке `left` значение функции отрицательно, а в точке `right` – положительно. Можно запускать двоичный поиск.

Последний тонкий момент – условие цикла двоичного поиска. Теперь значения в границах у нас не целые, поэтому останавливаться при разнице один или меньше рановато, погрешность ответа будет слишком большой. Вещественные числа представляются внутри через мантиссу и порядок, поэтому два соседних с точки зрения программы вещественных числа могут различаться на любую, в том числе большую величину. Если порядок чисел больше десяти в сотой, то соседние числа могут различаться уже на единицы, а то и на десятки, поэтому не стоит менять в условии единицу на какое-то маленькое число. Рекомендуем заменить цикл `while` на цикл `for` и просто проделать некоторое число итераций – например, пятьдесят. На каждой итерации диапазон поиска делится вдвое, поэтому в конце он уменьшится в $2^{50} \approx 1.12 \cdot 10^{15}$ раз. С большой вероятностью вам хватит такого количества итераций, хотя, конечно, всё зависит от ограничений задачи, и для более точной оценки в более сложных задачах стоит математически оценить число этих самый итераций цикла.

```python
for it in range(50):
    mid = (left + right) / 2
    if f(mid) < 0:
        left = mid
    else:
        right = mid
print((left + right) / 2)
```