Двоичный поиск по ответу

Сегодня мы рассмотрим одно из самых частых применений двоичного поиска: двоичный поиск по ответу. Иногда бывают задачи, когда с помощью формул или другими способами ответ вычислить проблематично, а вот проверить, является ли некоторое значение ответом, довольно просто. Рассмотрим такую ситуацию на примере.

Задача

У вас есть одна копия документа, ваша задача — получить n копий. Для этого в вашем распоряжении есть два копировальных автомата. Первый создает копию за x минут, а второй — за y минут. Требуется определить, за какое наименьшее время вы сможете получить требуемые n копий?.

Ясно, что на первом этапе нам нужно получить еще одну копию, чтобы в дальнейшем запустить оба автомата одновременно. Для этого нужно выбрать автомат, который быстрее сделает копию, и воспользоваться им. Чтобы решить задачу дальше, нужно уже применить более сложные рассуждения. Если n небольшое, то можно было бы просто моделировать ситуацию – поддерживать время, через которое освободится каждый из копиров, каждый раз выбирать минимум, перезапускать освободившийся копир и считать полученные копии. Однако если n велико (например порядка миллиарда), то такое моделирование будет работать очень долго.

Эту задачу можно решить с помощью математики и вывода формул, но гораздо проще сделать это с помощью двоичного поиска по ответу. Давайте заметим и используем следующее свойство. Если некоторого времени T достаточно, то есть за время T мы можем сделать n копий документа, то и любое время, большее T, тоже подходит. Аналогичное рассуждение и для случая, если некоторое время t не подходит – тогда с уверенностью можно сказать, что любое время, меньшее t, тоже не подходит. Посмотрим на функцию от времени t, которая возвращает, возможно ли за это время сделать n копий. Эта функция сначала возвращает отрицательный ответ, а затем, начиная с какого-то t, начинает возвращать положительный ответ:

$$\underbrace{----------}_{\substack{\text{ответ}\\ \text{отрицательный}}}\underbrace{+++++++++++}_{\substack{\text{ответ}\\ \text{положительный}}}$$

Нам нужно найти минимальное время, за которое возможно получить требуемые копии. Это означает, что нам нужно найти самый первый (самый левый) плюс в нашей

последовательности. Давайте заведем два указателя, левый и правый. Левый будет всегда указывать на минус, а правый — на плюс (это будет инвариант, то есть свойство, которое никогда не нарушается). А дальше мы воспользуемся двоичным поиском, чтобы шаг за шагом уменьшать расстояние между этими указателями и в конце концов сделать так, чтобы левый указывал на самый правый минус, а правый — на самый левый плюс.

Давайте заведем переменные под наши указатели. Для этого нужно найти время, которого точно не хватит, и время, которого точно хватит. Точно не хватит -1 минуты (0 минут может хватить, если нам нужно сделать только одну копию, которая у нас уже есть). Чтобы найти время, которого точно хватит, можно взять самый быстрый копир и представить, что все копии мы делаем на нём. Нужно сделать $n-1$ минус одну копию, каждая делается $\min(x, y)$ минут, то есть максимум нам понадобится $(n-1)\min(x, y)$ минут:

```
left = -1
right = (n-1)*min(x, y)
```

Теперь давайте будем двоичным поиском уменьшать расстояние между `left` и `right`. Будем это делать, пока они не указывают на соседние значения, то есть пока разница между `left` и `right` больше единицы. Дальше выбираем середину отрезка (`m = (left + right) // 2`) и определяем, подходит ли нам соответствующее время. Для этого напишем функцию `check(t)`, которая будет возвращать нам `True` или `False` в зависимости от того, успеем ли мы за время `t` сделать `n` копий. Если успеваем, то сдвинем правую границу, ведь в позиции `m` стоит плюс. Иначе в этой позиции стоит минус, сдвинем левую границу:

```
left = -1
right = (n-1)*min(x, y)
while right - left > 1:
    mid = (left + right) // 2
    if check(mid, n, x, y):
        right = mid
    else:
        left = mid
print(right)
```

Ответ в итоге будет равен `right`, ведь `right` указывает на подходящее значение, а оно минимально. `right - 1`, равный `left`, указывает на последнее неподходящее значение.

Это довольно универсальный шаблон для задач на двоичный поиск по ответу. Главное — четко зафиксировать инвариант для каждой из границ двоичного поиска, правильно подобрать стартовые значения и не ошибиться в функции `check`. Реализуем функцию `check` для нашей задачи. Нам нужно проверить, можем ли мы за `t` минут сделать `n` копий на копирах, создающих копию за `x` и `y` минут соответственно. Сначала сделаем вторую копию, а затем посчитаем, сколько копий мы сделаем за оставшееся время на

каждом из копиров. Если суммарно мы сделаем больше либо равно `n` копий, то время `t` нам подходит, иначе не подходит:

```
def check(t, n, x, y):
    t = t - min(x, y)
    n -= 2
    return t // x + t // n >= n
```

Рассмотримв еще один пример задачи на двоичный поиск по ответу.

Задача

У вас есть n прямоугольных дипломов размера $a \times b$. Вы хотите повесить их на квадратную доску наименьшего размера так, чтобы их все было видно, то есть чтобы они никак не накладывались. Поворачивать дипломы нельзя.

Для того, чтобы в задаче можно было использовать двоичный поиск по ответу, нужно убедиться в выполнимости ключевого свойства: если для квадрата размера x можно разместить все дипломы на доске, то и для любого квадрата большего размера тоже можно. Действительно, интуитивно это очевидно, ведь квадрат меньшего размера всегда включается в квадрат большего размера. Значит, мы можем воспользоваться двоичным поиском по ответу.

Определим инварианты на границы. Левая снова будет указывать на размер доски, на которой нельзя разместить n дипломов, а правая — на размер доски, на которой их разместить можно. Ответ снова будет в правом указателе. В качестве `left` можно сохранить `0`, на пустой доске ничего нельзя разместить, а в качестве `right` сохранить доску размера n, умноженное на максимальную из сторон: на такой доске мы точно можем разместить в ряд все дипломы.

Основная часть двоичного поиска по ответу никак не отличается от предыдущей задачи, нам все так же нужно выбирать середину и двигать границы в зависимости от результата функции `check`:

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

Помимо задания стартовых значений границ, изменится функция `check`. Теперь нам нужно посмотреть, сколько дипломов будет влезать по вертикали, а сколько по горизонтали. Если у нас зафиксирована длина стороны квадрата `x`, то размещать дипломы

нужно плотно, чтобы не оставалось свободного места между ними, как показано на картинке:

По горизонтали у нас будет влезать `x // a`, а по вертикали — `x // b`. Произведение этих величин будет давать общее число дипломов, помещающихся на квадратную доску:

```
def check(x, n, a, b):
    return (x // a)*(x // b) >= n
```

Часто бывают задачи, где ответ в итоге хранится в левой границе, то есть нужно найти максимальное значение, при котором некоторое условие выполняется. Функция тогда выглядит как последовательность плюсов, а затем последовательность минусов. Функция `check` иногда может быть сложнее, использовать больше переменных, списки, циклы и так далее, но от этого эффективность двоичного поиска не уменьшается. Позже мы посмотрим на более сложные применения двоичного поиска, точнее на более сложные функции проверки значения.