Find First Entry in List with Duplicates

In this lesson, you will learn how to find the first entry in a list with duplicates using a binary search in Python.

In this lesson, we will be writing a function that takes an array of sorted integers and a key and returns the index of the first occurrence of that key from the array.

For example, for the array:

```
[-14, -10, 2, 108, 108, 243, 285, 285, 285, 401]
```

with

```
target = 108
```

the algorithm would return `3`, as the first occurrence of `108` in the above array is located at index `3`.

The most naive approach to solving this problem is to loop through each element in the array. If you stumble upon the target element, it will be the first occurrence because the array is sorted. Otherwise, if the number does not exist in the array, we return `None` to indicate that the number is not present in a list.

```
def find(A,target):
  for i in range(len(A)):
    if A[i] == target:
      return i
    return None
```

The above code will work, but it will take linear time to complete as the time it takes for the code to execute is proportional to the size of the array for the worst-case.

Let's go ahead and apply a binary search to make our lives easier to solve this problem. We can reduce the problem from linear Big $O(n)$ to $O(logn)$ where n is the size of the array. In our current problem, we have non-distinct entries in sorted order, so we need to tweak the binary search algorithm to solve this variance of the problem.

Now we will tweak the binary search so that we can redefine the high point of the array based on whether the entry to the left of the target element is the same or not.

Implementation

Check out the code below:

```
def find(A, target):
    low = 0
    high = len(A) - 1

    while low <= high:
        mid = (low + high) // 2

        if A[mid] < target:
            low = mid + 1
        elif A[mid] > target:
            high = mid - 1
        else:
            if mid - 1 < 0:
                return mid
            if A[mid - 1] != target:
                return mid
            high = mid - 1

A = [-14, -10, 2, 108, 108, 243, 285, 285, 285, 401]
target = 108
x = find(A, target)
print(x)
```

Explanation

The code that we have written above is almost the same as the original binary search algorithm with some slight variation in statements on **lines 12-17**. The binary search will help us find the target element, but our problem is to find the first occurrence of that target element. The code above updates the lower and upper bounds for our search space according to the result of the comparison between the value of `target` and the value of the midpoint (`A[mid]`) (**lines 8 -11**) just as in the original Binary Search. The execution jumps to **line 13** when `target` is equal to `A[mid]`. Since the array is sorted in ascending order, our target is either the middle element or an element on its left. Now we have to make sure that we return the *first occurrence* from the function.

On **line 13**, we check if `mid - 1` is less than `0`, this is an edge case we would deal with if the *first occurrence* is on the first index. Next, we check if the element to the left of `A[mid]`, i.e., `A[mid - 1]` is the target element or not (**line 15**). If it isn't, the condition on **line 15** is `True`, and this implies that `A[mid]` is the *first occurrence*, so we return `mid` on **line 16**. However, if the element to the left is also the same as the target element, we update `high` to `mid - 1`. In this way, we condense our search space to find the *first occurrence* of the target which will be to the left of the midpoint.

One other way of thinking is to combine the linear approach and the binary search approach to solving this problem. You find your target element using binary search, and once you find it, you can keep going to the left of the array until you hit an element where the left of that element is not the element that we're looking for. This approach will work, but in the worst case you can have an element or an array consisting of all elements of the same number, and thus this approach will boil down to giving you the same runtime as the initial naive approach that we came up with previously.

I hope you are clear with everything we have learned so far. In the next lesson, we will have a look at Python's Bisect method. Stay tuned!