Binary Search

Key ideas:

Binary search also called bisection search is an algorithms to find an element quickly in a sorted array

① Is the element you want to find in the left half or the right half? => Right half.

② Repeat.

③ Finally there is just one element left. That's our element.

At each step you rule out half of the search space. How can the idea be so simple, but the implementations be so difficult?

It is not. You just have to think about it in the right way.

① The first thing we should decide on, is what you want to return. The object itself or the index where it is at? The index itself is a more useful piece of information. You can always just grab the object at that index later.

② What do we do when the object we are looking for is not there? Return -1? Raise exception? What if there are multiple valid indices to choose from? With every arbitrary choice we make, we would have to remember the implementation details that go with that choice.

A better way to do it is to rephrase the question, so that there is always one answer: If I were going to add another element, say 7, where should I put it so that it is the first 7. If there is no 7 in the array, the answer would be the index where you would put 7 to maintain the sorted property of the array. If there is at least one 7, the answer would be the index of the first 7.

- The index would be zero if the array is empty, which is not actually a valid index
- It could also be (n-1)+1 if 7 is absent from the array and greater than all elements in a non-empty array. This is also not a valid index.

The array being sorted is not actually the property being used in binary search. The property is that everything greater than 7 is to the right of 7 and everything less than 7 is to the left of 7.

Here, look, I changed some of the elements:

[2, 3, 5, 4, 6, 7, 9, 8]

and the array is no longer in sorted order but notice the steps are exactly the same. What we need is that everything less than 7 is to the left of 7 and everything greater than 7 should be to the right. In this array, the numbers have been replaced with the important information: Is this number less than 7?

[2, 3, 5, 4, 6, 7, 9, 8]

[ ] #<7?

Notice how any Trues are on the left and any Falses are on the right.

Now that we are looking at the relevant data, where do we insert 7 so that it is the first 7?

Notice that the answer we are looking for is the index of the first False value or the end of the array if there are no False values.

This is the key insight, we need to keep in mind that makes the implementation extremely easy to remember:

Squint your eyes and find the first False value.

OK, let's write the implementation:

```
def bisect(arr: list, x) -> int:
    lo = 0
    hi = len(arr)
    while lo < hi:
        mid = (lo + hi) // 2
        if arr[mid] < x:
            lo = mid + 1
        else:
            hi = mid
    return lo
```

lo and hi are lower and upper bounds on where this first False index is. 0 would mean that the first value is False and len(arr) would mean that all the values are False and we would have to put the new element at the end of the array.

The idea is that we are continuously decrease the upper bound and increase the lower bound until they are equal, lo and hi will meet at the answer index we are looking for.

So we could return either lo or hi, it makes no difference. We compute mid which is the closest index to the middle between lo and hi.

Python can represent arbitrarily large integers, so you do not have to worry about overflow. But, if you were using another language besides Python, you might have to worry about overflow,

In that case you could represent the midpoint equivalently by writing it as lo + half the distance between lo and hi. If you write it this way, it will never overflow.

```
# mid = (lo + hi) // 2
mid = lo + (hi - lo) // 2
```

Next comes the part of the algorithm that is often written incorrectly. We need to compare whatever is there in the array at the midpoint with our given x and then update our lower and upper bounds.

Remember, we are looking for the first False value where T or F is determined by the question is this (strictly) less than x.

```
if arr[mid] < x:
    lo = mid + 1
else:
    hi = mid
```

If the mid value is less than x; It is a True value.

Since the value at mid is a True value, the earliest a false value could occur is the next one, which is mid+1.

What about if we see a false value?

In this case, the first false value could not be any index before mid, because mid is a false value. But it could be mid or earlier than mid, there could be other false values before the one that we found. That means, the best we can do is update our upper bound to be whatever mid is.

And there you have it. This is a correct implementation.

But how do we know, that they always gives the correct answer.

We do know that lo is always less than or equal to the correct answer is always less than or equal to hi and we do keep updating the lower and upper bounds, but how do we know that they actually meet, so that the loop stops.

It is enough to show that on every iteration, the difference between lo and hi always decreases by at least one.

```
1.  def bisect(arr: list, x) -> int:
2.      lo = 0
3.      hi = len(arr)
4.      while lo < hi:
5.          mid = (lo + hi) // 2
6.          if arr[mid] < x: # True
7.              lo = mid + 1
8.          else:
9.              hi = mid
```

If we end up in clause on line 4, 5

lo = mid + 1

ensures that we are increasing the lower bound by at least one.

In clause 8, 9 we are setting hi to mid. So, hopefully we are lowering hi. But how do we know mid is actually strictly less than hi.

Here we are actually using the property that integer division rounds down. Inside the while loop: while lo < hi

lo is strictly less than hi.

So lo + hi is strictly less than hi + hi.

That means for calculating mid:

$$mid = (lo + hi) // 2.$$

When we divide (lo + hi) by 2 and round down we will get something strictly less than (hi + hi) // 2 i.e. hi.

Based Right

But wait. You ask: instead of returning a 7 so that it is the first seven, could not 2 returns the index, so that it could be the last 7?

You could. You could follow the same analysis and get a very similar algorithm.

[2, 3, 5, 4, 6, 7, 7, 7, 8, 9]

[T, T, T, T, T, F, F, F, F, F] # < 7

[F, T, T, T, T, T, T, T, F, F] # <= 7

To find where it should go to be the first 7, the relevant piece of information is: "is this (current element) strictly less than 7".

To find where it should go to be the last element, the relevant piece of information is

"Is this (current element) less than or Equal to 7". In either case, you are looking for the first false value. You just need to decide what you mean by true or false.

Bsed Right will return to insert position of index s.t. 7 is the last 7.

when True means arr[mid] < x

→ This is bsed left

bsed left will give you the insert position of 7 is the first seven.

when True means arr[mid] <= x

→ This is bsed right.