Binary search Python Why do we use mid + 1 or mid - 1

Asked 2 years, 1 month ago Active 2 years, 1 month ago Viewed 562 times

0

1

I am learning binary search and the sample code uses" `low = mid +1 and high = mid -1` " but I don't understand why we do not use " `low = mid and high = mid` " instead?

```
def binarysearch(sequence, value):
    lo, hi = 0, len(sequence) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if sequence[mid] < value:
            lo = mid + 1
        elif value < sequence[mid]:
            hi = mid - 1
        else:
            return mid
    return None

my_list = [1, 3, 5, 7, 9]
binarysearch(my_list, 3)
```

python data-structures binary-search

edited Nov 13 '17 at 4:07
Dinidu Hewage
1,864 6 32 41

asked Nov 13 '17 at 3:23
Mila
31 2

2 Answers

1

The reason is to avoid overlapping searches; it places the boundary of the search ranges on the items that have not been checked yet.

For instance: if mid is at index 10, the next search left will look at values up to index 9, and the right search from values at index 11.

```
                     mid
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
|                 |    |                           |
<-from 0 to mid-1->   <-- from mid+1 to the end -->

note that in this example, the boundary values are included
```

Reblochon Masque
23.9k 7 29 51

Hi, can you please give me an example? – Mila Nov 13 '17 at 3:27

Hi, your explanation helps me very well to understand that python will discard the half part that has been checked, including the boudaries. However, I do not understand that how it will help with the search, since Python will check the mid, not the boudaries. Can you please explain this to me? – Mila Nov 13 '17 at 3:48

By checking the mid, the search is reducing the search space by half each time. If you reduce the rearch range by 1 at each end each time, you are converging faster towards finding the value under consideration. – Reblochon Masque Nov 13 '17 at 4:06

0

Please let me try to explain why the binary search works. Let's say we've the following sequence `A=[-5, 10, 14, 33, 42, 42, 42]` and `searched_value = 14` we would have:

```
Iteration 1,     lo = 0, hi = 6, mid = 3     A[mid] > 14
Iteration 2,     lo = 0, hi = 2, mid = 1     A[mid] < 14
Iteration 3,     lo = 2, hi = 2, mid = 2     A[mid] = 14  (found!)
```

At each iteration of the algorithm we could conclude that `lo` and `hi` always contains the position of the searched value, and we'll prove it by induction on the number of iterations:

Inductive Hypothesis: if the searched value is present in the sequence it will always be contained between `lo` and `hi` .

Base Case: At iteration 1 `lo = 0` and `hi = n-1` contains all elements, in consequence if the searched value is present in the sequence it'll be contained between `lo` and `hi` , and the invariant is trivially correct.

Inductive Step: At any step if the searched value is contained between `lo` and `hi` , it'll continue to be contained between `lo` and `hi` at the next iteration. We've 3 possibilities (and here is the answer to the question):

- If `A[mid] = searched_value` : In this case the algorithm finishes reporting correctly the position of the searched value in the sequence and the invariant is correct because the searched value was between `lo` and `hi` .
- If `A[mid] < searched_value` : Knowing that it's a sorted sequence, all the elements between `A[lo...mid] < searched_value` (including `A[mid]`), thus we can assign `lo=mid+1` (safely searching only in the upper half) and the invariant will still be correct at the next iteration.
- If `A[mid] > searched_value` : knowing that it's a sorted sequence, all the elements between `A[mid...hi] > searched value` (including `A[mid]`), thus we can assing `hi=mid-1` (safely

Given that, at each iteration, the algorithm always searches on smaller segments of the sequence, the termination condition is guaranteed because either there's only 1 element that is `searched_value` or it is different and, at the next iteration, the algorithm is going to report that such element isn't present in the sequence.

In consequence the algorithm is proved correct (and the reason why we use `mid+1` and `mid-1`).

answered Nov 13 '17 at 4:59

Wilfredo
1,392 1 6 8