University of Toronto - Computer Science - Image

Searching for a Value in a List: Binary Search

Linear Search searches for a value in a list, starting at index `0`, working towards the end of the list. If the value is not contained in the list, then linear search will examine all items in the list. Linear search can be applied to both sorted and unsorted lists. However, if the list is sorted, then a faster searching algorithm, called binary search, may be used.

Binary Search

The binary search algorithm is used to find a value in a sorted list. To understand binary search, let us look at the following diagram:

items less than v	v	items greater than v

Let us assume that there are 100,000 items in the above list. The middle item, let us call it `v`, is compared to the value that we are looking for. Since the list is sorted, every item to the left of `v` is less than or equal to `v`, and the items to the right of `v` are greater than or equal to `v`. If the value we are looking for is greater than `v`, then we can quickly deduce that it is not to the left of `v`, and thus we eliminate 50,000 items of the list after making just one comparison.

Implementation

For our implementation, we need to understand a few invariants. At all times, our list is divided into three regions. On the left, we have all the items that are less than `v`. On the right, we have all the items that are greater than or equal to `v`. In the middle, we have all the items that are still unknown. We will mark the beginning of the unknown section with `b`, and the end with `e`. This will be known as the binary loop invariant:

	b	e
items less than v	unknown items	items greater than or equal to v

In the beginning, we know nothing about all the items in the list. Therefore, the initial state has `b` located at the beginning of the list, and `e` located at the end of the list:

b e

unknown items

The code for our initial state is:

```
b = 0
e = len(L) - 1
```

At the end of this process, the unknown section is empty. Everything to the right is greater than or equal to `v` and everything to the left is less than `v`. Index `b` is to the right of index `e`. This how our list looks at the end:

items less than v	e	b	items less than v

Here is the code for the binary search function:

```
def binary_search(L, v):
    """ (list, object) -> int

    Precondition: L is sorted from smallest to largest, and
    all the items in L can be compared to v.

    Return the index of the first occurrence of v in L, or
    return -1 if v is not in L.

    >>> binary_search([2, 3, 5, 7], 2)
    0
    >>> binary_search([2, 3, 5, 5], 5)
    2
    >>> binary_search([2, 3, 5, 7], 8)
    -1
    """

    b = 0
    e = len(L) - 1

    while b <= e:
        m = (b + e) // 2
        if L[m] < v:
            b = m + 1
        else:
            e = m - 1

    if b == len(L) or L[b] != v:
        return -1
    else:
        return b
```

In the above function, the main `while` loop continues to loop as long as `b` is less than `e`. Once `b` is greater than `e`, than our search is complete. Inside the loop, we first set `m` to the middle index between index `b` and index `e`. Inside the `while` loop, there is an `if` statement. If the item at position `m` is less than `v`, then we advance `b` to be the index just to the right of `m`. However, if the item at index `m` is greater than `v`, then we decrease `e` so that it is the index to the left of `m`.

Finally, at the end of our function, we have one last `if` statement. When the `while` loop ends, there are two reasons why it might have ended. First, if all the items in the list are less than `v`, then `b` will end up being equal to the length of the list, and will refer to an index outside of the list; we check whether `b` is equal to the length of `L`, and if it is, then we return `-1`. Finally, we check the item at location `b`. After all, it is possible that `v` is not contained in the list. If `L[b]` is equal to `v`, then we return `b`. Otherwise we will return `-1`.

Jennifer Campbell • Paul Gries
University of Toronto
