

# Unit 23: Searching

## Learning Objectives

After this unit students should be familiar with linear search and binary search algorithm, including understanding how the algorithms work, implementing the algorithms, arguing their correctness, and analyzing their running time.

### Linear Search

Let's continue the discussion on efficiency on one of the fundamental problems in computing: how to look for something.

You have seen a similar problem in the problem Index in Exercise 3: Given a list of items  $L$  and query item  $q$ , find if  $q$  is in  $L$ .

Let's write a function to solve this. Our function returns  $-1$  if  $q$  is not in the list. It returns the position of  $q$  in the list otherwise.

```

1 long search(long n, const long list[], long q) {
2     for (long i = 0; i < n; i += 1) {
3         if (list[i] == q) {
4             return i;
5         }
6     }
7 }
```

What is the **worst-case running time**, expressed in Big-O notation, of the function above? Suppose the query  $q$  is not in the list, we will have to scan the whole list, once. The worst case running time is, therefore,  $O(n)$ .

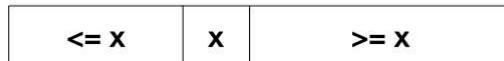
Can we do better? It turns out that this running time  $O(n)$  is also the best that we can do because we cannot be sure that  $q$  does not exist until we check every single element in the list. So there is no shortcut.

**Binary Search** ↗ if there are repeated values, must see the algo to see which index it will return  
- because depends on where the chop happens (depends on how mid is)

But, do we always have to check every element in the list? It turns out that, like many real-life situations, **if the input list is sorted**, we do not have to scan through every element. We

can eliminate a huge chunk of the elements based on whether a chosen element is bigger or smaller than  $q$ .

Suppose that the input list is sorted in increasing order. Pick a random element  $x$  from the list. Any element to the left of  $x$  must be less than or equal to  $x$ , and any element to the right of  $x$  must be greater or equal to  $x$ .



Suppose that  $q > x$ , then we know that  $q$  must be to the right of  $x$ , there is no need to search to the left of  $x$ . Otherwise,  $q < x$ , and  $q$  must be to the left of  $x$ , and there is no need to search to the right of  $x$ .

We can code this out using a recursive function below:

```

1  /**
2   * Look for q in list[i]..list[j].
3   *
4   * @pre list is sorted
5   * @return -1 if not found, the position of q in list otherwise.
6   */
7  long search(const long list[], long i, long j, long q) {
8      if (i > j) { // base case where i>j
9          return -1; // means something happened and
10     } // not found
11     long mid = (i+j)/2;
12     if (list[mid] == q) {
13         return mid;
14     }
15     if (list[mid] > q) {
16         return search(list, i, mid-1, q); // searching lower half
17     }
18     return search(list, mid+1, j, q); // else search upper half
19 }
```

start the search from middle to eliminate the most at a time

searching between i,j indexes

We call this function with:

```
1 | search(list, 0, n-1, q);
```

The search algorithm above is called *binary search* since it repeatedly cut the range of values to search by half.

## Why is it correct?

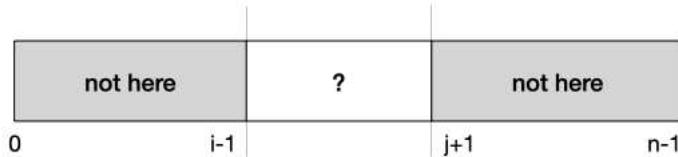
It is not obvious at first glance that the code above correctly searches for  $q$  in  $L$ .

Let's analyze this function more systematically by writing an assertion for this function.

What we want to do here is to eliminate elements in the array that cannot possibly contain  $q$  -- these are elements outside of the `list[i] .. list[j]` range. In other words, we want **to assert that**

```
1 | // { q is not in list[0]..list[i-1] and list[j+1]..list[n-1] }
```

at the beginning of the function. **In other words, this is a precondition for the function.**



Let's see if this precondition is true at the beginning. Since  $i$  is 0 and  $j$  is  $n - 1$ , the ranges `list[0]..list[i-1]` and `list[j+1]..list[n-1]` are empty, so the assertion is true.

**how it reaches the base case** || What happens if  $i > j$ ? This implies that  $i - 1 > j - 1$ , so the range `list[0]..list[i-1]` and the range `list[j+1]..list[n-1]` overlap. We can be sure that  $q$  is not anywhere in `list`.

Let's see how we ensure this assertion is true in the recursive call.

```
1 |     :
2 |     if (list[mid] > q) {
3 |         return search(list, i, mid-1, q);
4 |     }
5 |     :
```

Line 3 of the snippet above is invoked only if `list[mid] > q`. Since the array `list` is sorted, we know for sure that any element in `list[mid+1]..list[j]` is larger than  $q$ . So,  $q$  cannot be anywhere in that range. We can assert, between Line 3 and Line 4 above:

```
1 | // { q is not in list[0]..list[i-1] and list[mid]..list[n-1] }
```

Thus, when Line 4 is invoked, the same assertion holds true. You can apply the same argument to the call:

```
1     return search(list, mid+1, j, q);
```

To summarize, we annotate the code above with the assertions:

```
1  /**
2   * Look for q in list[i]..list[j].
3   *
4   * @pre list is sorted
5   * @pre q is not in list[0]..list[i-1] and list[j+1]..list[n-1]
6   * @return -1 if not found, the position of q in list otherwise.
7   */
8  long search(const long list[], long i, long j, long q) {
9      if (i > j) {
10         // { q is not in list[0]..list[n-1] }
11         return -1;
12     }
13     long mid = (i+j)/2;
14     if (list[mid] == q) {
15         return mid;
16     }
17     if (list[mid] > q) {
18         // { q is not in list[0]..list[i-1] and list[mid]..list[n-1] }
19         return search(list, i, mid-1, q);
20     }
21     // { q is not in list[0]..list[mid] and list[j]..list[n-1] }
22     return search(list, mid+1, j, q);
23 }
```

## How Efficient is Binary Search

We have seen that if the input list is not sorted, then we minimally have to check every element in the list, leading to an  $O(n)$  algorithm.

With a sorted input and using binary search, however, we can do a better. Let's consider the worst case, where  $q$  is not in the list. Note that every comparison we make, we reduce the range of elements to search by half, until we reach one element. We start with  $n$  elements that could possibly contain  $q$ . After one comparison, we are left with  $n/2$ . After another comparison, we are left with  $n/4$ , etc. It takes only  $O(\log_2 n)$  steps until we reach one element in the list. This is a big improvement over  $O(n)$  time: Suppose  $n$  is 1,000,000,000,000,000. Instead of scanning through and comparing one quadrillion elements, we only need to compare 50 of them!

## Problem Set 23

## Problem 23.1

(a) ~~Rewrite the code for binary search above using a loop instead of recursion.~~

(b) Instead of returning the position of the query `q`, modify ~~the binary search algorithm in~~  
~~(a)~~ an iterative version of the binary search such that it returns either:

- a position `k`, such as `a[k] <= q <= a[k+1]`.
- `-1` if `q < a[0]`
- `n-1` if `q > a[n-1]`