Comparing Search Algorithms

We have seen two ways to search for a value in a list. Linear search works on both unsorted and sorted lists. Binary search works only on sorted lists.

Let's take a deeper look at each of the searching algorithms.

Comparing Linear Search and Binary Search

A Deeper Look at Linear Search

Let us consider the following example in order to understand the number of steps linear search will take to complete:

1	2	3	4	5

Let us look for the value `6`, which does not occur in the list. We compare `6` to the item at index `0`, which is `1`, and since this is not the value we are looking for, we will move on to the next item. Next, we compare `6` to `2`. Again, we move on. We will compare `6` to `3`, then `4`, and then `5`. So, for a list of 5 items, we examined all 5 items in the list.

Let us make a table of the number of iterations we make as the number of items in the list grows:

# items	# iters
1	1
2	2
3	3
.	.
.	.
.	.

A Deeper Look at Binary Search

We know that for a list with one item, the number of iterations is also one. However, what happens when there are two items? Well, with the first comparison, we eliminate half of the list, and we are left with one item. This leaves us with one more comparison. So, for a list with two items, the number of iterations is two.

Now, let us consider what happens when our list has 4 items: With just one comparison, we are able to eliminate half the list, and thus, we are left with only two items. At this point, we have only two items left, and we know from our previous calculation that we will need two more iterations for that. Therefore, for a list of 4 items, we only needed 3 iterations. Let us now construct a table with this information:

# items	# iters
1	1
2	2
4	3
8	4
16	5
32	6

64	7
128	8

This is very interesting! If a list has 128 items to begin with, then after only one iteration, we will only have to consider 64 items. This is very powerful. With one interation of the loop, we are able to eliminate half the list. This is only possible because the list is sorted.

Logarithms

$\log_2(n)$ is the logarithm base 2 of n, and we can think of that as the number of times we divide n by 2 in order to reach 1. Here is quick table of values and their logarithms of base 2:

n	log_2(n)
2	1
4	2
8	3
16	4
32	5

Time to Run Linear and Binary Search

Number of iterations

In Python, `math.log(x, b)` returns the logarithm of `x` to the base `b`.

```
>>> math.log(2, 2)
1.0
>>> math.log(4, 2)
2.0
>>> math.log(8, 2)
3.0
>>> math.log(16, 2)
4.0
>>> math.log(32, 2)
5.0
>>> math.log(1000000000, 2)
29.897352853986263
```

These calculations can give us an indication of how many iterations binary search will take. Since we know that binary search eliminates half the list on every iteration, we can use `math.log(x , b)` to predict the number of iterations it will take for a list with `x` items. So, for a list with 1 billion items (1,000,000,000), binary search will perform about 30 iterations. On the other hand, linear search will take about 1,000,000,000 iterations in the worst case.

Profiling

The `cProfile` module allows us to *profile* a piece of code. By profile, we mean to measure how much time it takes to run, and how much memory it uses. Here is how we can time our searching functions using a module named `cProfile`:

```
>>> import cProfile
>>> L = list(range(10000000))
>>> len(L)
10000000
>>> cProfile.run('binary_search(L, 10000000)')
         5 function calls in 0.000 seconds

   Ordered by: standard name
```

```
    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
         1    0.000    0.000    0.000    0.000 :1(binary_search)
         1    0.000    0.000    0.000    0.000 :1()
         2    0.000    0.000    0.000    0.000 {len}
         1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
>>> cProfile.run('linear_search(L, 10000000)')
          10000005 function calls in 9.146 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    6.198    6.198    9.146    9.146 :1(linear_search)
        1    0.000    0.000    9.146    9.146 :1()
 10000002    2.948    0.000    2.948    0.000 {len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Looking at the above code, `cProfile` is able to give us a nice breakdown of the function calls involved in each searching algorithm. We see that all function calls take 0.000 seconds in binary search. However, in linear search, the total time spent is more than 9 seconds, and we notice that the function call to `len` happens 10,000,002 times. This presents an opportunity for us to save time. In our linear search algorithm, we can save a huge amount of time by storing the length of the list into a local variable. The new and faster linear search algorithm is shown below, as well as its running time:

```
>>> def linear_search(L, v):
      """ (list, object) -> int
      Return the index of the first occurrence of v in L,
      or return -1 if v is not in L.
      """
      i = 0

      length = len(L)

      while i != length and v != L[i]:
          i = i + 1

      if i == length:
          return -1
      else:
          return i6
>>> cProfile.run('linear_search(L, 10000000)')
         4 function calls in 1.872 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.872    1.872    1.872    1.872 :1(linear_search)
        1    0.000    0.000    1.872    1.872 :1()
        1    0.000    0.000    0.000    0.000 {len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

In the above profile, we see that `len` is called only once, as opposed to 10,000,002 times in the previous version of `linear_search(L, v)`. Also, the total time of linear search dropped down to about 2 seconds in this faster implementation of linear search.

Jennifer Campbell • Paul Gries
University of Toronto
