Computer Science I — CSCI-141
Search — Lecture (2/2)

09/22/2015

1 Problem

Given a list, L, of N sorted elements, and a target value, T, can you devise an algorithm that finds the index of T in L, without having to examine each element? In other words, does an algorithm exist that at worst performs better than linear search when searching for an element in a sorted list? If the program cannot find the target element it should indicate so.

Here is an example of such a program in action:

```
Step 1 - Create your sorted data...
Start: 10
Stop: 22
Step: 1

Data:  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
Number of elements:  12

Step 2 - Enter target value to search for...
Target: 11

Step 3 - Get index of target value in data (None if doesn't exist)...
Searching for 11:  [10, 11, 12, 13, 14] *15* [16, 17, 18, 19, 20, 21]
Searching for 11:  [10, 11] *12* [13, 14]
Searching for 11:  [] *10* [11]
Searching for 11:  [] *11* []

11 found at index 1
```

2 Analysis and Solution Design

The solution to this problem is to use *a divide and conquer algorithm*. Here is an outline of the approach to seach for a target in a list:

1. If there are no more elements to look at then the target is not present.
2. Compute the index of the middle element and use it to access the value.
3. If the target is equal to the element then return the index.
4. If the target is less than the element then the target must be in the lower half of the list. Re-run from step 1 and consider only the lower half of the list.
5. If the target is greater than the element than the target must be in the upper half of the list. Re-run from step 1 and consider only the upper half of the list.

We can write this algorithm either iteratively or recursively. We will consider only the recursive implementation because the algorithm emerges so easily from the steps above.

2.1 Integer Division

Python version 3 has an operator for integer division to calculate index values.

```
print( 6 // 4 )   # double slash is Python3's integer division
1
```

2.2 Algorithm and Implementation

```
def binary_search( data, target, start, end ):
    """
    binary_search :
      List(Orderable) Orderable NatNum NatNum -> NatNum or None
    Perform binary search for target between start and end indices.
    Returns: index of target in data, if present; otherwise None.
    pre-condition: the data collection is already sorted.
    """

    # base condition - terminate when start passes end index
    if start > end:
        return None    # sentinel value indicates failure

    # find the middle value between start and end indices
    mid_index = ( start + end ) // 2
    mid_value = data[mid_index]

    if target == mid_value:
        return mid_index
    elif target < mid_value:
        return binary_search( data, target, start, mid_index-1 )
    else:
        return binary_search( data, target, mid_index+1, end )

def get_index( data, target ):
    """
    get_index : List(Orderable) Orderable -> NatNum or None
    get_index returns the index of target in data or None if
    not target found.
    Parameters:
        data - a list of sorted data
        target - the target value to search for
    Returns:
        The index of the target element in data, if it is present,
        otherwise None.
    """

    # search for the target across all elements in data
    return binary_search( data, target, 0, len( data ) - 1 )
```

3 Execution Diagram and Substitution Trace

For all function invocations, the `data` argument is [10, 20, 30, 40, 50, 60, 70, 80].

3.1 Execution Diagram

```
binary_search( data, target:=50, start:=0, end:=7 )
    # start <= end
    mid_index = 3
    mid_value = 40
    # target > mid_value
    binary_search( data, target:=50, start:=4, end:=7 )
        # start <= end
        mid_index = 5
        mid_value = 60
        # target < mid_value
        binary_search( data, target:=50, start:=4, end:=4 )
            # start <= end
            mid_index = 4
            mid_value = 50
            # target == mid_value
            return mid_index
```

3.2 Substitution Trace

```
binary_search( data, 50, 0, 7)  =  binary_search( data, 50, 4, 7)
                                =  binary_search( data, 50, 4, 4)
                                =  4
```

4 Testing (Test Cases, Procedures, etc.)

Let's look at some scenarios to confirm why the overall performance of this algorithm is guaranteed to be better than linear time complexity, $O(N)$.

First, we consider the best case scenario. This happens when the target value is the first element sought; the target is the middle element in the list. It took only one operation since we needed to look at *only one entry* of the collection to find the value.

```
L = [ 10, 20, 30, 40, 50, 60, 70, 80 ]
T = 40
Searching for 40: [10, 20, 30] *40* [50, 60, 70, 80]
40 found at index 3
```

Now let's consider a worst case scenario for a list of $N == 8$. If we look at the first divide and conquer operation, it produces 3 elements in the lower half and 4 elements in the upper half. Let's search for an element in the upper half to see what happens.

The next divide and conquer will leave one element, 50, in the lower half and two elements, [70, 80], in the upper half (of the upper half). Since this dividing pattern will continue for the 'upper, upper, upper' half, if we search for the last element in the original list as the target value, we will experience the worst case performance for a successful search. That performance, shown below, is 4 searches for $N == 8$.

```
    L = [ 10, 20, 30, 40, 50, 60, 70, 80 ]
    T = 80
    Searching for 80: [10, 20, 30] *40* [50, 60, 70, 80]
    Searching for 80: [50] *60* [70, 80]
    Searching for 80: [] *70* [80]
    Searching for 80: [] *80* []
    80 found at index 7
```

What if we search for an element that is not in the list? The algorithm should return a value of `None` as a *sentinel* indicating the target is not in the list. If the element is larger than the last element in the list, a search count of 4 is also the worst case for $N == 8$.

```
    L = [ 10, 20, 30, 40, 50, 60, 70, 80 ]
    T = 85
    Searching for 85: [10, 20, 30] *40* [50, 60, 70, 80]
    Searching for 85: [50] *60* [70, 80]
    Searching for 85: [] *70* [80]
    Searching for 85: [] *80* []
    85 is not in the list.
```

Now let's consider a list with 16 elements. The maximum number of searches is 5.

```
L = \
 [10, 20, 30, 40, 50, 60, 70] *80* [90, 100, 110, 120, 130, 140, 150, 160]
T = 160
Searching for 160 : \
 [10, 20, 30, 40, 50, 60, 70] *80* [90, 100, 110, 120, 130, 140, 150, 160]
Searching for 160 : [90, 100, 110] *120* [130, 140, 150, 160]
Searching for 160 : [130] *140* [150, 160]
Searching for 160 : [] *150* [160]
Searching for 160 : [] *160* []
160 found at index 15
```

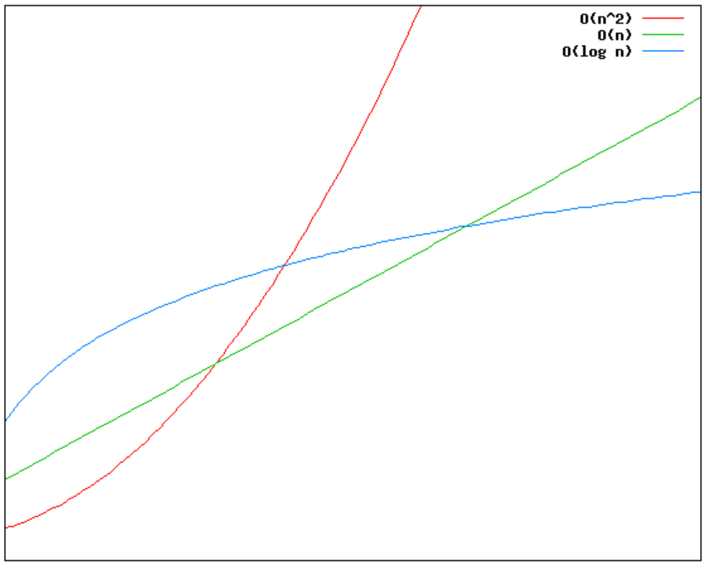
Here is a list of the maximum number of searches based on list size, N:

```
  N = 4, max searches = 3
  N = 8, max searches = 4
  N = 16, max searches = 5
  N = 32, max searches = 6
```

This yields the formula, $maxSearches = \log_2(N) + 1$.

Based on this data we conclude that the time complexity of binary search is a logarithm of the data size, $O(\log N)$, where the base of the logarithm is two.

A *binary search* divides and conquers to search more quickly than a linear search.



The graph above displays the growth curves for the three categorizations of big-O that we have covered so far. Notice how $O(\log N)$ grows more slowly than $O(N)$, and how $O(N)$ in turn grows more slowly than $O(N^2)$.

Now we can list test cases for our function:

1. Search an empty list for any target value:
```
binary_search( [], 10, 0, -1 ) -> None
```

2. Search a list of one element for a target value that is in the list, a target less than any element in the list, *and* a target greater than any element in the list:
```
binary_search( [10], 10, 0, 0 ) -> 0
binary_search( [10], 5, 0, 0 ) -> None
binary_search( [10], 15, 0, 0 ) -> None
```

3. Search a larger list for a target value that is in the list, a target less than any element in the list, a target greater than any element in the list, *and* a target value that is *not* in the list but within the range:
```
binary_search( [10, 20, 30, 40, 50], 30, 0, 4 ) -> 2
binary_search( [10, 20, 30, 40, 50], 5, 0, 4 ) -> None
binary_search( [10, 20, 30, 40, 50], 60, 0, 4 ) -> None
binary_search( [10, 20, 30, 40, 50], 25, 0, 4 ) -> None
binary_search( [10, 20, 30, 40, 50], 45, 0, 4 ) -> None
```

Tests of even numbered lists are unnecessary; recursion has done them.