CMSC 351: Binary Search

Justin Wyss-Gallifent

September 25, 2023

1 What it Does

Given a sorted list of elements and a target element, finds the index of the target element or returns failure if the target element does not exist.

2 How it Works

The algorithm first looks at the middle of the list. If if element is not there then it knows by comparison if the element is on the left or the right of that middle element and so it concentrates its search to half the list and repeats. It keeps repeating this process either until it finds the element or the sublist it is looking at shrinks to length 1 and the element is not found.

Example 2.1. Consider the list with 20 elements. We wish to find the number 17. We look at the entire list:

$$A = [\underline{0, 0, 4, 4, 6, 7, 8, 9, 9, 10, 12, 13, 13, 14, 14, 17, 18, 19, 19, 19}]$$

We reference the start and end by indices so we have $L = 0$ and $R = 19$. We find the center $C = \lfloor (19 + 0)/2 \rfloor = 9$ and find $A[9] = 10$. This is too small so 17 must be to the right.

We check the sublist by reassigning $L = 9 + 1 = 10$ and leaving $R = 19$:

$$[0, 0, 4, 4, 6, 7, 8, 9, 9, 10, \underline{12, 13, 13, 14, 14, 17, 18, 19, 19, 19}]$$

We find the center $C = \lfloor (19 + 10)/2 \rfloor = 14$ and find $A[14] = 14$. This is too small so 17 must be to the right.

We check the sublist by reassigning $L = 14 + 1 = 15$ and leaving $R = 19$:

$$[0, 0, 4, 4, 6, 7, 8, 9, 9, 10, 12, 13, 13, 14, 14, \underline{17, 18, 19, 19, 19}]$$

We find the center $C = \lfloor (19 + 15)/2 \rfloor = 17$ and find $A[17] = 19$. This is too large so 17 must be to the left.

We check the sublist by leaving $L = 15$ and reassigning $R = 17 - 1 = 16$:

$$[0, 0, 4, 4, 6, 7, 8, 9, 9, 10, 12, 13, 13, 14, 14, \underline{17, 18}, 19, 19, 19]$$

We find the center $C = \lfloor (16 + 15)/2 \rfloor = 15$ and find $A[15] = 17$. This is exactly right so we return 15.

3 Pseudocode

In the following pseudocode we assign time values not to get a precise time measurement but only to look at the complexity. This is why we have not separated the time values for conditional checks versus bodies.

```
\\ PRE: A is a sorted list of length n.
\\ PRE: TARGET is a target element.
function binarysearch(A,TARGET)
    L = 0                                   } c1
    R = n-1                                 }
    while L <= R                            ??? iterations
        C = floor((L+R)/2)                  }
        if A[C] == TARGET                   }
            return C                        }
        elif TARGET < A[C]                  } c2
            R = C-1                         }
        elif TARGET > A[C]                  }
            L = C+1                         }
        end                                 }
    end while
    return FAIL                             c3
end
\\ POST: Value returned is either the index or FAIL.
```

Note 3.0.1. A note to make sure that this does as intended in the `FAIL` case:

Assign `L=0` and `R=n-1`. Take the middle index `C=floor((L+R)/2)`. and examine `A(C)`. If `A(C)==TARGET` then we return `C`. Otherwise if `TARGET<A(C)` then `TARGET` is to the left of `C` so we set `R=C-1` and start again. On the other hand if `TARGET>A(C)` then `TARGET` is to the right of `C` so we set `L=C+1` and start again.

This process proceeds until either when we find `TARGET` or when `L=R` and `TARGET` does not exist. In this latter case what happens is that we assign `C=floor((L+R)/2)=L=R` and then since `A(C)!=TARGET` we end up with either $L = C + 1 = R + 1$ or $R = C - 1 = L - 1$ and in both cases $R < L$.

4 Time Complexity Analysis

1. **Best-Case:**

   If the target is immediately located at `C=floor((L+R)/2)` at the start of the first iteration then the total time requirement is:

$$T(n) = c_1 + c_2 = \Theta(1)$$

2. **Worst Case:**

   The worst-case scenario happens if the `TARGET` is never found.

   Consider the length of the list after each iteration. Initially it is length n.

   After the first iteration the sublist length is $\frac{n}{2}$. Technically if n is odd then the length is this value ± 0.5 but this doesn't affect what follows.

   After the second iteration the sublist length is $\frac{n}{4}$.

   This continues such that after k iterations the sublist length is $\frac{n}{2^k}$.

   In a worst case scenario the `while` loop iterates until `L==R` and then at the end of that iteration `R<L` and it fails. We have `L==R` when the sublist length is 1:

$$\begin{aligned}\frac{n}{2^k} &= 1\\ 2^k &= n\\ k &= \lg n\end{aligned}$$

   However noting that it has one more iteration when `L==R` we can then conclude that the loop iterates $1 + \lg n$ times and then the condition fails.

   It follows that the total time requirement is:

$$T(n) = c_1 + c_2(1 + \lg n) + c_3 = \Theta(\lg n)$$

3. **Average Case:**

   An average case can be defined by examining all possible positions of the `TARGET` within the list and taking the average time requirement assuming all possible positions are equally likely.

   The calculation will be easier if we look at the case where $n = 2^N - 1$ for $N \in \mathbb{Z}^+$. Let's analyze the number of iterations of the while loop required for each element in the list.

   When $n = 2^1 - 1 = 1$ there is only one element and it is found after one iteration. We could put this in a really boring table:

# elements	# iterations
1	1

   When $n = 2^2 - 1 = 3$ there is one element which is found after one iteration and now let's stop to make an observation. The remaining elements will be in a left or right sublist and each of these sublists has length 1. It follows that effectively, one iteration later, we have two copies of the $N = 1$ case, meaning we will have two copies of that case with one more iteration each:

# elements	# iterations
1	1
2(1)=2	1+1 = 2

When $n = 2^3 - 1 = 7$ there is one element which is found after one iteration and then the remaining elements will be in a left or right sublist each of which is a copy of the $N = 2$ case so we'll have two copies of that case with one more iteration each:

# elements	# iterations
1	1
2(1)=2	1+1=2
2(2)=4	2+1=3

And again for $n = 2^4 - 1 = 15$:

# elements	# iterations
1	1
2(1)=2	1+1 = 2
2(2)=4	2+1 = 3
2(4)=8	3+1 = 4

In general when $n = 2^N - 1$ we have the following:

# elements	# iterations
1	1
2	2
4	3
8	4
$\vdots$	$\vdots$
2^{N-1}	N

Now then, the probability of getting an element requiring some number of iterations equals the number of such elements divided by n, and i iterations takes $c_1 + ic_2$ time, so we can write the following table:

Probability	Time
$\frac{1}{n}$	$c_1 + 1c_2$
$\frac{2}{n}$	$c_1 + 2c_2$
$\frac{4}{n}$	$c_1 + 3c_2$
$\frac{8}{n}$	$c_1 + 4c_2$
$\vdots$	$\vdots$
$\frac{2^{N-1}}{n}$	$c_1 + Nc_2$

The expected time is then:

$$\begin{aligned}
\sum_{i=1}^{N} (\text{Probability})(\text{Time}) &= \sum_{i=1}^{N} \frac{2^{i-1}}{n} \left[c_1 + ic_2\right] \\
&= \frac{1}{n} \sum_{i=1}^{N} 2^{i-1} \left[c_1 + ic_2\right] \\
&= \frac{1}{n} \left[c_1 \sum_{i=1}^{N} 2^{i-1} + c_2 \sum_{i=1}^{N} i 2^{i-1} \right] \\
&= ... = \Theta(\lg n)
\end{aligned}$$

5 Thoughts, Problems, Ideas

1. Show the steps of binary search when looking for the value 17 in the list $\{-3, 4, 7, 17, 20, 30, 40, 51, 105, 760\}$. At each step give the value of C and how the comparisons update L and R. Inside the `while` loop how many comparisons are made?
2. How does the particular pseudocode in the notes behave if the target exists at multiple indices?
3. Adjust the pseudocode so that if the target exists at multiple indices the function returns the first occurrence. Find the $\mathcal{O}$ worst-case time complexity of this pseudocode.
4. Adjust the pseudocode so that if the target exists at multiple indices the function returns the last occurrence. Find the $\mathcal{O}$ worst-case time complexity of this pseudocode.
5. Assuming the list values are distinct, adjust the pseudocode so that if the target does not exist the function returns the smallest value larger than the target.
6. Explain how your answer to the previous question can be used to modify InsertSort. Pseudocode is not necessary, a good explanation will suffice.
7. In the pseudocode implementation in the notes the worst-case total time requirement is $T_B(n) = c_1 + c_2(1 + \lg n)$. A straightforward linear search is linear, something like $T_L(n) = c_3 + c_4 n$. If $c_1 = 2$, $c_2 = 10$ (it's a big compound statement), $c_3 = 1$ and $c_4 = 2$, Plot these functions together on an axis. for which n will each be faster? This question is intended to be computer-assisted.
8. Suppose the sorted list `A` is infinitely long. In other words think of `A` as an increasing function defined for all integers $n \geq 0$. Here some ideas for extending binary search:
   - Start with `L=0` (of course) and `R` set at some arbitrary positive integer.
   - If we have ever found some `C` with `TARGET < A[C]` then we can basically just do binary search on `A[0,...,C]`.
   - If we have never found some `C` with `TARGET < A[C]` then each time we encounter `TARGET > A[C]` we double R and continue.

   (a) Show how modifying the algorithm this way and using `R=4` and `TARGET=15` will work with the list:

   $$A = [0, 4, 5, 10, 11, 12, 15, 16, 18, 20, 30, 31, 50, 100, 117, 118, 119, 200, 203, ...]$$

   (b) Write the pseudocode for this algorithm.

9. Suppose that due to some noise in the data exactly one of the comparisons will register as incorrect.

   (a) Give a specific example to illustrate that Binary Search could completely fail.

   (b) Give a specific eample to illustrate that Binary Search might still work.

10. Suppose the two-dimensional array `A` of size $n \times m$ which is indexed as `A[0,...,n-1][0,...,m-1]` contains integers with the property that every entry is greater than or equal to the entry directly above it and is greater than or equal to the entry directly to the left of it.

    An example of such an array is:

$$\begin{bmatrix} 2 & 7 & 8 & 10 & 14 \\ 3 & 8 & 9 & 11 & 15 \\ 6 & 10 & 12 & 12 & 20 \\ 7 & 11 & 13 & 13 & 21 \\ 8 & 11 & 15 & 20 & 22 \end{bmatrix}$$

    Write the pseudocode for a function `binarysearch2d` which locates a target entry in the array and returns the location. Your algorithm should use a 2D version of binary search.

    Hint: In the above example the middle entry is 12. If the target is less than 12 where could it be? If the target is more than 12 where could it be?

11. Binary Search only works on a sorted list. Suppose we have an unsorted list A and we wish to check if an element is in the list. We could sort it using one of our three sorts and then check if Binary Search returns `FAIL` or not. From a time perspective why is this a bad choice?

6 Python Test

Code:

```
import random
import math
def binarysearch(A,TARGET):
    L = 0
    R = len(A)-1
    while L <= R:
        print('Checking: '+str(A[L:R+1]))
        C = math.floor((L+R)/2)
        if A[C] == TARGET:
            print('Checking: [' + str(A[C]) + ']')
            return(C)
        elif TARGET < A[C]:
            R = C-1
        elif TARGET > A[C]:
            L = C+1
    return(False)
A = []
for i in range(0,20):
    A.append(random.randint(0,50))
A.sort()
print(A)
TARGET = 10
result = binarysearch(A,TARGET)
if result == False:
    print('Not Found')
else:
    print('Found at index ' + str(result))
```

Output:

```
[1, 2, 7, 10, 12, 14, 15, 16, 17, 17, 22, 23, 28, 30, 37, 43, 45, 46, 49,
Checking: [1, 2, 7, 10, 12, 14, 15, 16, 17, 17, 22, 23, 28, 30, 37, 43, 45
Checking: [1, 2, 7, 10, 12, 14, 15, 16, 17]
Checking: [1, 2, 7, 10]
Checking: [7, 10]
Checking: [10]
Checking: [10]
Found at index 3
```

And:

```
[3, 3, 3, 5, 5, 6, 7, 8, 9, 15, 20, 22, 26, 28, 28, 29, 30, 32, 39, 49]
Checking: [3, 3, 3, 5, 5, 6, 7, 8, 9, 15, 20, 22, 26, 28, 28, 29, 30, 32,
Checking: [3, 3, 3, 5, 5, 6, 7, 8, 9]
Checking: [6, 7, 8, 9]
Checking: [8, 9]
Checking: [9]
Not Found
```

Fun fact related to `A.sort()`: Python's default `sort` method uses Timsort. Timsort is a merge sort and insertion sort hybrid which works well on real-world data in which there are often *runs* of sorted sublists within the list. Timsort essentially collects and merges those runs. Timsort is $\mathcal{O}(n \lg n)$.