CMSC 351: Binary Search

Justin Wyss-Gallifent

February 23, 2021

1 What it Does .................................................. 2
2 How it Works .................................................. 2
3 Pseudocode and Time Complexity ...................... 3
4 Thoughts, Problems, Ideas ............................... 6
5 Python Test .................................................. 8
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 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:

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

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 \( R = 19 \):

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

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

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

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

We find the center \( C = \lfloor (19 + 15)/2 \rfloor = 17 \) and find \( A[17] = 17 \). This is exactly right so we return 15.
3  Pseudocode and Time Complexity

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.

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

**Note 3.0.1.** A note to make sure that this does as intended in the \text{FAIL} case: Assign \L=0 and \R=n-1. Take the middle index \C=\text{floor}((\L+\R)/2). and examine \text{A(C)}. If \text{A(C)}==\text{TARGET} then we return \text{C}. Otherwise if \text{TARGET}<\text{A(C)} then \text{TARGET} is to the left of \text{C} so we set \R=\text{C}-1 and start again. On the other hand if \text{TARGET}>\text{A(C)} then \text{TARGET} is to the right of \text{C} so we set \L=\text{C}+1 and start again.

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

1. **Best-Case:**

   If the target is immediately located at \text{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 \( \pm 0.5 \) 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:

   \[
   \frac{n}{2^k} = 1 \\
   2^k = n \\
   k = \log_2 n
   \]

   Thus the loop iterates \( \log n \) times and then the condition fails.

   It follows that the total time requirement is:

   \[
   T(n) = c_1 + c_2 \log n + c_3 = \Theta(\log n)
   \]

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 cleanest examples to look at are where \( n = 2^N - 1 \) for some \( N \) since all of these result in divisions which never require the floor function. Here are some examples:

   - When \( n = 3 = 2^2 - 1 \) there are:
     - 1 element in the middle which has time requirement \( c_1 + c_2(1) \) and probability \( \frac{1}{3} \).
     - 2 elements at the ends which each have time requirement \( c_1 + c_2(2) \) and each has probability \( \frac{1}{3} \) for a total probability of \( \frac{2}{3} \).

   - When \( n = 7 = 2^3 - 1 \) there are:
     - 1 entry in the middle which has time requirement \( c_1 + c_2(1) \) and probability \( \frac{1}{7} \).

   The remaining are recursive versions of the above applied to the two sides with an additional iteration (and hence larger time requirement) and with lower probability, yielding:
- 2 elements in the two middles each have time requirement $c_1 + c_2(2)$ and each has probability $\frac{1}{7}$ for a total probability of $\frac{2}{7}$.
- 4 elements at the four ends each have time requirement $c_1 + c_2(3)$ and each has probability $\frac{1}{7}$ for a total probability of $\frac{4}{7}$.

In general we have the pattern, for $n = 2^N - 1$:

- 1 element with time $c_1 + c_2(1)$ and probability $\frac{1}{n}$.
- 2 elements each with time requirement $c_1 + c_2(2)$ and probability $\frac{1}{n}$ for a total probability of $\frac{2}{n}$.
- 4 elements each with time requirement $c_1 + c_2(3)$ and probability $\frac{1}{n}$ for a total probability of $\frac{4}{n}$.
- ...and so on until...
- $2^{N-1}$ elements each with time requirement $c_1+c_2(N)$ and probability $\frac{1}{n}$ for a total probability $\frac{2^{N-1}}{n}$.

Observe that we have some idea that are counting correctly because:

$$1 + 2 + 4 + ... + 2^{N-1} = 2^N - 1 = n$$

This tells us that we have the correct number of elements and that the total probability is 1.

The expected time requirement is then:

$$\left( \frac{1}{n} \right) (c_1 + 1c_2) + \left( \frac{2}{n} \right) (c_1 + 2c_2) + ... + \left( \frac{2^{N-1}}{n} \right) (c_1 + Nc_2)$$

$$= \frac{1}{n} \left[ c_1 \sum_{i=0}^{N-1} 2^i + c_2 \sum_{i=1}^{N} i2^{i-1} \right]$$

$$= \frac{1}{n} \left[ c_1 (2^N - 2) + \frac{1}{2} c_2 \left( N2^{N+1} - 2^{N+1} + 2 \right) \right]$$

$$= \frac{1}{n} \left[ c_1 (n+1 - 2) + \frac{1}{2} c_2 \left( \lg(n+1)(n+1) - 2(n+1) + 2 \right) \right]$$

$$= \frac{1}{n} \left[ c_1(n - 1) + c_2(n + 1) \lg(n + 1) - n \right]$$

$$= \Theta(\lg n)$$
4 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 \( T_L(n) = c_3 + c_4n \). 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 \( \text{TARGET} < A[C] \) then we can basically just do binary search on \( A[0, \ldots, C] \).
   - If we have never found some \( C \) with \( \text{TARGET} < A[C] \) then each time we encounter \( \text{TARGET} > A[C] \) we double \( R \) and continue.

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

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

   (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 example to illustrate that Binary Search might still work.

10. Suppose the two-dimensional array $A$ of size $n \times m$ which is indexed as $A[0,\ldots,n-1][0,\ldots,m-1]$ 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 $\text{binarysearch2d}$ 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 $\text{FAIL}$ or not. From a time perspective why is this a bad choice?
5 Python Test

Code:

```python
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, 22, 23, 28, 30, 37, 43, 45, 46, 49, 50]
Checking: [1, 2, 7, 10, 12, 14, 15, 16, 17, 22, 23, 28, 30, 37, 43, 45, 46, 49, 50]
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, 29, 30, 32, 39, 49]
Checking: [3, 3, 3, 5, 5, 6, 7, 8, 9, 15, 20, 22, 26, 28, 29, 30, 32, 39, 49]
Checking: [3, 3, 3, 5, 5, 6, 7, 8, 9, 15, 20, 22, 26, 28, 29, 30, 32, 39, 49]
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 $O(n \log n)$. 