Classical Searching Algorithms and Correctness

Piseth Tang

September 30, 2024

1 Introduction

We begin with the definition of our problem, in particular, the searching problem which can be stated as follows:

Searching Problem Definition

Input: A sequence [a] of n numbers $(a_0, a_1, ..., a_{n-1})$ stored in an array and a key x.
Output: An index i such that $x = A[i]$ or the special value $None$ or (NIL in other languages) if x does not appear in A.

[a] In this case, sequence refers to the one in mathematics, therefore there is an ordering, from $\mathbb{N}$ imposed on the array which stores the sequence itself. To put it simply, any elements can be unambiguously compared.

2 Intuition behind each algorithm

2.1 Naïve Linear Search

The simplest search algorithm is the **Naïve Linear Search**. In this algorithm, we start from the beginning of the array and examine each element one by one. If an element matches the target, we return the index. Otherwise, if we reach the end of the array without finding the target, we conclude that the element is not present. The search continues linearly through the array.

Example: Consider the array $A = [5, 12, 9, 3, 21, 4]$ and the target $x = 9$. The algorithm proceeds as follows:

- Compare 5 with 9 $\rightarrow$ no match.
- Compare 12 with 9 $\rightarrow$ no match.
- Compare 9 with 9 $\rightarrow$ match found at index 2.

2.2 Dichotomic Binary Search (Recursive)

The **Binary Search (Recursive)** algorithm works on sorted arrays. It repeatedly divides the search interval in half. If the target value is less than the middle element of the array, the search continues in the left half, otherwise in

the right half. The process is repeated recursively until the element is found or the interval is empty.

Example: Consider the sorted array $A = [1, 3, 7, 9, 15, 20, 25]$ and the target $x = 15$:

- Initial array: middle element is 9.
- $15 > 9$, so search continues in the right half: $[15, 20, 25]$.
- Middle element is 20.
- $15 < 20$, so search continues in the left half: $[15]$.
- $15 = 15$, element found.

2.3 Dichotomic Binary Search (Iterative)

The **Binary Search (Iterative)** follows the same principle as the recursive version but avoids recursion by using a loop. It uses two pointers, low and high, to represent the current search range. The search interval is halved until the target is found or the interval becomes empty.

Example: Consider the same sorted array $A = [1, 3, 7, 9, 15, 20, 25]$ and the target $x = 15$:

- Start with low $= 0$ and high $= 6$, middle element is 9.
- $15 > 9$, so update low to 4.
- Middle element is 20.
- $15 < 20$, so update high to 4.
- Middle element is 15, element found.

3 Implementation for each algorithm

3.1 Insertion Sort

```python
def naive_search_position_with_bounds(t, e, start, end) -> tuple:
    comps = 0
    for i in range(start, end):
        comps += 1
        if t[i] == e: return i, comps  # position found!
    return None, comps  # for all i, no position found!
def naive_search_position(t, e):
    return naive_search_position_with_bounds(t, e, 0, len(t))
```

Listing 1: Naïve Linear Search

```python
def dichotomic_search_position_with_bounds(t, x, begin, end):
    if end - begin == 1:  # base case
        if x == t[begin]:
            return begin
        return None
    # general case
    middle = (begin + end) // 2
```

```
    if x == t[middle]:  # exact position found!
        return middle
    if x < t[middle]:  # on the left...
        return dichotomic_search_position_with_bounds(t, x, begin,
    middle)
    return dichotomic_search_position_with_bounds(t, x, middle, end
    )
def dichotomic_search_position(a, e):
    return dichotomic_search_position_with_bounds(a, e, 0, len(a))
```

Listing 2: Dichotomic Binary Search (Recursive)

```
def iterative_dichotomic_with_comps(arr, key):
    begin, end = 0, len(arr)
    comps = 0
    while begin < end:
        mid = (begin + end)//2
        comps += 1
        if arr[mid] == key: return mid, comps
        elif arr[mid]  > key: end = mid
        else: begin = mid + 1
    return None, comps # key wasn't found.
```

Listing 3: Dichotomic Binary Search (Iterative)

Just like any diligent students, we have also tested the above implementations. We test their consistency over N sorted arrays $\{A_{1\leq s\leq N}\}, \forall s \in [1, N], \text{card}\,(A_k) = s$. For each A_k, we tested consistency for all integer values in $[A_k[0]-1, A_k[-1]+1]$. Note that the notation $A[-1]$ denotes the last element in an array in *Python*. The reader can verify access the entire codes here [1].

4 Result

As expected, the naïve linear search cannot measure up to its more efficient divide and occur variants. Right from the onset, dichotomic search algorithms make less number of comparisons and as the size of the array grows, the gap widens exponentially!

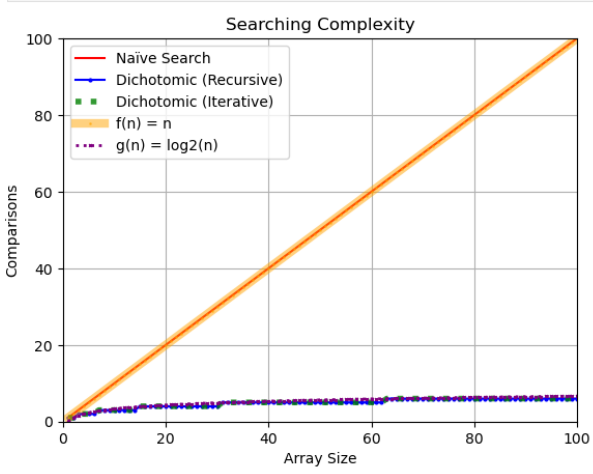

Figure 1: Average Number of Time

However, we did not include this here, but one could ask how much space does each algorithm require to execute from start to finish? Then, this is when stack-based recursive algorithm like our Dichotomic recursive variant may fall short. Then, we could say, "Does that our Dichotomic iterative algorithm works best in **all cases**?", not exactly. It depends on the application, and when it comes to theoretical analysis like this one that we are carrying out, we can only go so far as to provide the results, but the solution will depend on the application or problem at hand. For instance, suppose we are working on a database that entails a lot of search queries and that such operation will happen **frequently** throughout the day, then it'd be wise to employ the iterative Dichotomic approach, as more searches will smooth out the initial caveats of sorting. However, if the opposite is true about our database, then it would be wise to stick with the linear search algorithm.
Having said all of that, we will dive into the *correctness* of each algorithm.

5 Correctness using invariant theory

A **loop invariant** [7] is a formalism used to prove the correctness of algorithms involving loops. More concretely, it is a property of a program; or in logic, a statement, that holds true before and after each iteration of a loop.
Typically, a proof of correctness using a loop invariant requires us to define three steps:

1. **Initialization**: Show that the invariant is true before the first iteration of the loop.
2. **Maintenance**: Prove that if the invariant holds before an iteration of the loop, it remains true before the next iteration.
3. **Termination**: Show that when the loop terminates, the invariant (along with the loop's exit condition) implies that the algorithm has correctly solved the problem.

A perceptive math student would be able to tell that a a loop-invariant proof bears similarity to another kind of proof called Loop invariants are commonly used in the correctness proofs of iterative algorithms, but can also be applied to recursive algorithms by considering each recursive call as a loop-like step (in fact, every recursive program can be converted to to an iterative one given sufficient amount of memory - that is to say it is theoretically possible but not always practically since unbounded memory is only theoreticall possible [1]). For recursive algorithms, we use a variant of the loop invariant called a **recursion invariant**, which must hold at every recursive call and termination. For further reading on loop invariants and their application in algorithm analysis, see [[4], [5], [6]].

For the proofs below, we will use natural language to get the ideas across. In most cases, one will see mathematical symbols such as $\forall, \in, \notin,$ etc. being used for efficiency. We will avoid that in this document as this is geared towards the general audience. Before we state the invariant (the statement that we want to prove), we will present the algorithm again in pseudocode so that the readers could see and verify why each proof makes sense.

[1] Those who are interested may have a look at [[2], [3]]

5.1 Linear Search's Correctness

Algorithm 1 Naïve Search

```
1: function NAÏVE-SEARCH(A, n, x)
2:     for i = 0 to n − 1 do
3:         if A[i] = x then
4:             return i                          ▷ Element found at index i
5:         end if
6:     end for
7:     return None                               ▷ Element not found
8: end function
```

We define the following loop invariant for linear search:

Linear Search Loop Invariant

At the start of the i-th iteration of the loop, the subarray $A[0, \ldots, i-1]$ does not contain the target element x.

Proof. **Initialization**: Before the first iteration (when $i = 0$), no elements have been checked, so the subarray $A[0, \ldots, i-1]$ is empty. Therefore, the invariant holds trivially.
Maintenance: At the start of the i-th iteration, assume that x is not present in the subarray $A[0, \ldots, i-1]$. The algorithm compares $A[i]$ with x:

- If $A[i] \neq x$, then our assumption holds, that is, the sub-array $A[0, ..., i-1]$ does not contain x, an now with the addition that $A[i] \neq x$ tells us that $A[1, ..., i]$ does not contain x, which exactly implies to us that at the start of the $(i+1)th$ iteration, such a condition holds.
- If $A[i] = x$, the algorithm terminates,as line 4 is executed and the index of x is returned.

Termination: The loop terminates for two reasons:

1. When $A[i] = v$ (line 4 is executed.)
2. When we are at the last test of the for loop, that is at the beginning of the $i = n$th iteration (in line 2 right before we jump out of the loop, it should be noted that the body of the loop **will not be** executed at this iteration.) Now at this line, the invariant holds as the sub-array $A[0, ..., n-1]$ (the entire array, in fact) does not contain the target x and the `None` value is returned, as desired.

Thus in both cases, $Naïve - Search(A, n, x)$ terminates. □

5.2 Iterative Dichotomic Search's Correctness

Algorithm 2 Binary Search (Iterative)

```
1: function BinarySearchIterative(A, n, x)
2:     low ← 1
3:     high ← n
4:     while low ≤ high do
5:         mid ← ⌊(low + high)/2⌋
6:         if A[mid] = x then
7:             return mid                    ▷ Element found at index mid
8:         else if A[mid] < x then
9:             low ← mid + 1
10:        else
11:            high ← mid − 1
12:        end if
13:    end while
14:    return −1                             ▷ Element not found
15: end function
```

Binary Search Loop Invariant

At the start of each iteration of the loop, the target element x is contained within the subarray $A[\text{low}, \ldots, \text{high}]$.

Proof. **Initialization**: Initially, low $= 0$ and high $= n - 1$, so the subarray is $A[0, \ldots, n-1]$, which is the entire array. Since x is either in the array or not, the invariant holds.
Maintenance: On each iteration, the algorithm checks the middle element $A[\text{mid}]$:

- If $A[\text{mid}] = x$, the algorithm terminates, and the search is successful.
- If $A[\text{mid}] > x$, the search continues in the left subarray $A[\text{low}, \ldots, \text{mid}-1]$, preserving the invariant.
- If $A[\text{mid}] < x$, the search continues in the right subarray $A[\text{mid}+1, \ldots, \text{high}]$, preserving the invariant.

Termination: The loop terminates when low $>$ high, meaning that the search range is empty. By the invariant, x is not present in the array, and the algorithm returns `None`. □

5.3 Recursive Dichotomic Search's Correctness

For recursive binary search, the concept of loop invariants can be applied using recursion invariants. The same invariant used for the iterative version applies here:

Algorithm 3 Binary Search (Recursive)

```
1: function BinarySearchRecursive(A, low, high, x)
2:     if low > high then
3:         return −1                                  ▷ Element not found
4:     end if
5:     mid ← ⌊(low + high)/2⌋
6:     if A[mid] = x then
7:         return mid                                 ▷ Element found at index mid
8:     else if A[mid] < x then
9:         return BinarySearchRecursive(A, mid + 1, high, x)
10:    else
11:        return BinarySearchRecursive(A, low, mid − 1, x)
12:    end if
13: end function
```

Binary Search Recursion Invariant

At the start of each recursive call, the target element x is contained within the subarray $A[\text{low}, \ldots, \text{high}]$.

Proof. **Initialization**: At the first recursive call, the sub-array is $A[0, \ldots, n-1]$, which is the entire array, so the invariant holds.
Maintenance: At each recursive call, the algorithm divides the sub-array and makes a recursive call to either the left or right half:

- If $A[\text{mid}] = x$, the search is successful, and the recursion ends.
- If $A[\text{mid}] > x$, the algorithm recurses on $A[\text{low}, \ldots, \text{mid} - 1]$.
- If $A[\text{mid}] < x$, the algorithm recurses on $A[\text{mid} + 1, \ldots, \text{high}]$.

The recursion invariant is maintained as long as the algorithm continues the search in the correct half.
Termination: The recursion ends when the sub-array is empty (low $>$ high), meaning that x is not present, and the algorithm returns `None`.

□

6 Summary

We summarize each of the searching algorithm listed above with its time complexity into a compact table below:

Algorithm	Best Case	Worst Case	Average Case
Naïve Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search (Recursive)	$O(1)$	$O(\log n)$	$O(\log n)$
Binary Search (Iterative)	$O(1)$	$O(\log n)$	$O(\log n)$

References

[1] Link to the code base.

[2] This cs thread contains helpful discussions on the topic of recursion and iteration.

[3] This MIT's page provides explicit examples on recursion.

[4] *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein (CLRS), Chapter 2: Loop Invariants.

[5] *Algorithm Design Manual* by Steven Skiena, Section 1.7: Loop Invariants.

[6] CMU Loop Invariants Lecture Notes: A concise resource on using loop invariants to prove algorithm correctness.

[7] As per usual, the Wikipedia page provides a somewhat good starting point for diving into any topic of the mathematical nature.