Binary Search: An Implementation Guide

Kunal Singh

Technical Report
Bhopal, MP, India
January 7, 2025

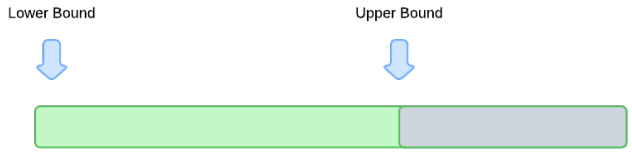
kunalsin9h@gmail.com

Abstract

Binary Search is on the most popular *searching* algorithms. It is used in variety of applications, but implementing it requires careful consideration of various components like *lower* and *upper* bounds,*mid* value, conditions and answers. We present a general method of **implementing** binary search. This process always give the correct result and it is very easy to work with. We have focused more on implementation then explaining different binary search concepts.

1 Introduction

Binary search is a highly efficient searching algorithm that operates on ***sorted collections*** of data, employing a ***divide-and-conquer*** strategy by systematically dividing the search space in half until the target element is found. The algorithm maintains two essential components: the `lower` and `upper` bounds, which define the current search space.



These boundaries are dynamically adjusted as the search progresses, with the lower bound typically starting at the first index and the upper bound at the last index of the array.

The algorithm's effectiveness relies on the **predicate function**, which evaluates positions within the search space and returns a boolean value to determine the target's location relative to the current position. This function must maintain monotonically to establish a clear true/false boundary across the search space.

Binary search achieves logarithmic time complexity `O(log n)` by eliminating half of the remaining search space in each iteration, making it particularly valuable in applications such as database index lookups, sorted array searching, and optimization problems where the search space can be represented as a sorted sequence.

An common implementation of binary search goes like:

Algorithm: Binary Search
1. **Input:** $key, array$
2. **Output:** $index$
3. $lower \leftarrow 0$
4. $upper \leftarrow len(array) - 1$
5. **while** $lower \leq upper$ **do**
6. $\quad mid \leftarrow (lower + upper)/2$
7. $\quad$ **if** $array[mid] = key$ **then**
8. $\quad\quad$ Return mid
9. $\quad$ **if** $array[mid] > key$ **then**
10. $\quad\quad upper \leftarrow mid - 1$
11. $\quad$ **else**
12. $\quad\quad lower \leftarrow mid + 1$
13. Return -1

While implementing Binary Search, we may face few problems, they are:

- What is the value of `lower bound`?
- What is the value of `upper bound`?
- How do I calculate my `mid` value?
- What condition should i write in my `while loop`?
- When my predicate is true, should i assign my `mid` to `lower` or `upper` bound?

- Whats my final answer, value of `lower` or `upper` bound?

These question are very common while implementing binary search, many times we do mistake and we have to do some debugging to land on correct implementation.

Lets see how we use the presented method for implementing binary search in a generalized manner to achieve consistency and correctness.

2 First True Binary Search

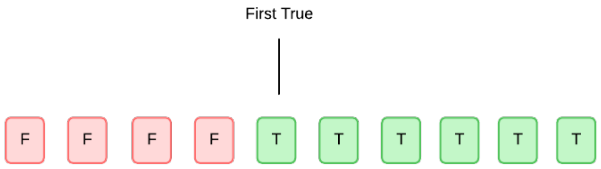
The first true binary search is a method for identifying the first element in a sorted list that evaluates to true when tested with a given predicate function.

Formally,

$$FirstTrue(A, P) = min\{x \in A : P(x) = true\}$$

Where:
A is sorted array / list
P is the predicate function
$min(...)$ finds the smallest element satisfying the condition

Graphically,



We present a generalized algorithm implementation method *first true binary search*, with changes from the classic implementation as:

1. Set `lower` bound value to be *minimum possible answer*, for an sorted array we can say 0 ($0th$ index).
2. Set `upper` bound value to be *maximum possible answer* + 1, for an sorted array we can say:

$$maxAnswer \leftarrow len(array) - 1$$
$$upper \leftarrow maxAnswer + 1$$
$$upper \leftarrow len(array)$$

3. Always use $lower < upper$ as **condition** in `while` loop.
4. Find mid value as normal:

$$mid \leftarrow (lower + upper)/2$$

5. If **predicate** function gives $true$ then set $high$ to mid else set $lower$ to $mid + 1$.
6. Answer is always the value of $lower$ bound.

The complete algorithm goes like:

Algorithm: Binary Search (First True)
1. **Input:** $key, array$
2. **Output:** $index$
3. $lowestPossibleAnswer \leftarrow 0$
4. $highestPossibleAnswer \leftarrow len(array) - 1$
3. $lower \leftarrow lowestPossibleAnswer$
4. $upper \leftarrow highestPossibleAnswer + 1$
5. **while** $low < high$ **do**
6. $\quad mid \leftarrow (low + high)/2$
7. $\quad$ **if** $P(array, key, mid) = true$ **then**
8. $\quad\quad upper \leftarrow mid$
11. $\quad$ **else**
12. $\quad\quad lower \leftarrow mid + 1$
13. Return $lower$

Where:
P is a predicate function, which returns $true/flase$, in case of sorted array, we can define function P as:

$$P(array, key, mid) = \begin{cases} \text{True} & \text{if } array[mid] \geq key \\ \text{False} & \text{otherwise} \end{cases}$$

Hence, following these rules we will always implement a correct binary search algorithm in case of first true.

3 Last True Binary Search

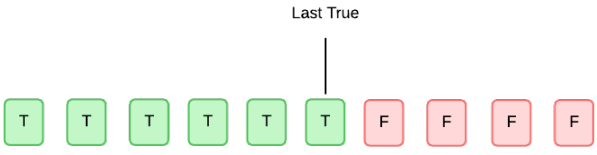
The last true binary search is a method for identifying the last element in a sorted list that evaluates to true when tested with a given predicate function.

Formally,

$$LastTrue(A, P) = max\{x \in A : P(x) = true\}$$

Where:
A is sorted array / list
P is the predicate function
$max(...)$ finds the largest element satisfying the condition

Graphically,



The generalized algorithm implementation method *first true binary search*, with changes from the classic implementation as:

1. Set `lower` bound value to be *minimum possible answer* - 1, for an sorted array we can say -1.

$$minAnswer \leftarrow 0$$
$$lower \leftarrow minAnswer - 1$$
$$lower \leftarrow -1$$

2. Set `upper` bound value to be *maximum possible answer*, for an sorted array we can say $len(array) - 1$:

3. Always use $lower < upper$ as **condition** in `while` loop.

4. Find mid value with adding 1:

$$mid \leftarrow (lower + upper + 1)/2$$

5. If **predicate** function gives $true$ then set low to mid else set $upper$ to $mid - 1$.

6. Answer is always the value of $lower$ bound.

The complete algorithm goes like:

Algorithm: Binary Search (Last True)
1. **Input:** $key, array$
2. **Output:** $index$
3. $lowestPossibleAnswer \leftarrow 0$
4. $highestPossibleAnswer \leftarrow len(array) - 1$
3. $lower \leftarrow lowestPossibleAnswer - 1$
4. $upper \leftarrow highestPossibleAnswer$
5. **while** $low < high$ **do**
6. $\quad mid \leftarrow (low + high + 1)/2$
7. $\quad$ **if** $P(array, key, mid) = true$ **then**
8. $\quad\quad lower \leftarrow mid$
11. $\quad$ **else**
12. $\quad\quad upper \leftarrow mid - 1$
13. Return $lower$

Where:

P is a predicate function, which returns $true/flase$, in case of sorted array, we can define function P as:

$$P(array, key, mid) = \begin{cases} \text{True} & \text{if } array[mid] < key \\ \text{False} & \text{otherwise} \end{cases}$$

Again, same as *first true*, following these rules we will always implement a correct binary search algorithm in case of last true.

4 The Conclusions

In conclusion, we can say following these methods for implementing will grantees us the correctness of algorithms. We no longer need to worry about the different components involved in binary search algorithm, and can solely focus on refining the **predicate** function. You can create a template for reusing it multiple times.

5 Future Work

There are two things I would like to explore in future, the generalization of binary search involving *floating point* precision. And writing **proof** of correctness for these methods.

6 Acknowledgment

This work is inspired from the code submitted by Benq (a top rated competitive programmer) on Codeforces. He has created templates for these methods. Here is the link to solutions.