A Survey of Binary Search

Brandon Ronald
#001313502
April 25, 2017

Introduction

In this paper we will compare two variants of Binary Search: The traditional variant known most commonly to programmers, and that of the general variant. We will see that there are in fact many advantages of using the general binary search algorithm, as it is both efficient and simple to remember.

Traditional Binary Search

The traditionally understood variant of binary search is defined as an algorithm that returns the index position of a targeted search value in a sorted array. This is accomplished by comparing the target value with the middle element in the array. If the target value is equal to the middle value, the index of the middle value is returned. If the target is larger than the middle element, the first half of the array is disregarded and a binary search is done on the second half of the array. Conversely, if the target is smaller, a binary search is done on the first half. [1]

Pseudocode Implementation

```
int traditionalBinarySearch(int F[], int N, int key){
    int low = 0;
    int high = N;
    while (low <= high) {
        int mid = low + ((high - low) / 2);
        if (F[mid] < key)
            low = mid + 1;
        else if (F[mid] > key)
            high = mid - 1;
        else
            return mid;
    }
    return - (low + 1);
}
```

General Binary Search

In the general variant of binary search, the input is an array or function F in which the first and last elements a and b are related to one another, by some co-transitive relation $\boldsymbol{Z}$. That is, a is $\boldsymbol{Z}$-related to b,

or *a* ***Z*** *b*. The goal is to find two neighbouring elements in *F* that are ***Z***-related to one another, if such a pair exists.

The algorithm begins by initializing variables `x` and `y` to the endpoints of *F, a* and *b* respectively. A loop is used, and the invariant is maintained: "`x` is ***Z***-related to `y`, and `x` is within the bounds of `F`". At each iteration, a median `m` is updated as the midpoint between `x` and `y`, that is, `m = (x+y)/2`. Because of the co-transitive property of ***Z***, at each iteration, either:

a. The midpoint is ***Z***-related to `y`, or
b. `x` is ***Z***-related to the midpoint.

The invariant `x` ***Z*** `y` is maintained by either setting `x` to `m` if the midpoint is ***Z***-related to `y`, or setting `y` to `m` if `x` is ***Z***-related to the midpoint . When `x` and `y` are neighbours `(x+1=y)`, `x` is returned.

Pseudocode Implementation

```
int generalBinarySearch(int F[], int N,  int key){
    int low = 0;
    int high = N;
    if (zRelated(key, F[low], F[high])){
        while(low+1!=high){
            int mid = (low+high)/2;
            if (zRelated(key, F[mid], F[high]))
                low = mid;
            else if (zRelated(key, F[low], F[mid]))
                high = mid;
         }
    }
    else return -1;
    return low;
}
```

For which we define our ***Z***-relation as appropriate for the problem. Here we will use a pseudocode example for computing the square root:

```
bool zRelated(int key, int x, int y){
    return (pow(x,2) <= key && key <= pow(y,2));
}
```

Remembering Binary Search

> *"Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky…" — Professor Donald Knuth*

Binary search is indeed a simple algorithm to understand and remember the basics of, however, when it comes to implementing correct code for the algorithm, the minute details become difficult to specify. For instance, even experienced programmers may forget how to set the low, high, and mid indices correctly at each iteration. There may also be confusion as to whether bounds should be exclusive or inclusive, or how

to correctly calculate the midpoint. Furthermore, overflow even becomes an issue of concern with arrays of large size. Lastly, there may be difficulty with how to properly implement the three-way result of the comparison between the value of the midpoint and the key.

Conversely, the general method does not require incrementing or decrementing any indices, or any three-way case implementation. The programmer must only remember to check at each loop where the midpoint fits in the $\boldsymbol{Z}$-relation, that is, if $m \; \boldsymbol{Z} \; y$ or $z \; \boldsymbol{Z} \; m$. This makes for a much more elegant, simple-to-remember solution.

Why do they look so different?

There are a few reasons why the two variants differ so much in shape. First, lets take a look at the loop guards:

```
low+1 != high    vs.    low <= high
```

The loop guards differ between the two because in traditional binary search, for the case in which the target value is not in the array, the `high` bound will be continuously decremented until it is lower than the `low` bound, at which point the loop terminates. For the general variant, the program always terminates the loop when `low` and `high` are neighbours, and the invariant does all the work to ensure that `low` is the appropriate return value.

The second difference is the presence of `+/- 1`'s in the body of the traditional version. For each iteration the loop in the traditional version there is 3 cases: The midpoint is either greater than, less than, or equal to the search key. If the midpoint is less than or equal to the key, the midpoint should be excluded in the next loop, therefore the new high or low bound must be incremented or decremented the by 1. In general binary search, maintaining the invariant only means that the new bounds of the search array are $\boldsymbol{Z}$-related to one another, and thus should be inclusive.

The final difference is seen when calculating the midpoint:

```
int mid = low + ((high - low) / 2)
                vs.
      int mid = (low+high)/2;
```

The difference here is that in the traditional variant, there is concern about overflow for arrays of large size. However, this method does not resolve overflow issues when non-conventional array indexing is used, and so in the general variant, the midpoint is calculated directly.

Myths about Binary Search

Some may argue that the traditional variant of binary search is better as it can exit the loop early via a **`return`** or **`break`** command. However, this is not true, and as we will see, in some cases the general variant will terminate after *less* iterations than the traditional variant. We will examine a simple pseudocode trace example. Consider an array of elements `a := { 1, 3, 4, 5, 7 }` and a search value `t:=6`. Let's look at a trace of the two respective binary search variants above for this input.

Traditional Variant

Initialization

low := 0
high := N-1 [= 4]

1st Iteration

check: low <= high [= true]
    mid := low + ((high - low) / 2) [= 2]
    check : F[mid] < key [= true]
        low := mid + 1 [= 3]
    skip

2nd Iteration

check: low <= high [= true]
    mid := low + ((high - low) / 2) [= 3]
    check: F[mid] < key [= true]
        low := mid + 1 [= 3]
    skip

3rd Iteration

check: low <= high [= true]
    mid := low + ((high - low) / 2) [= 4]
    check: F[mid] < key [= false]
        skip
    check: F[mid] > key [= true]
        high := mid - 1 [= 3]
    skip

4th Iteration

check: low <= high [= false]
    skip

The traditional binary search algorithm takes 3 full loop iterations, breaking at the beginning of the fourth iteration.

General Variant

Initialization

low := 0
high := N - 1 [= 4]

1st Iteration

check: low + 1 != high [= true]
    m = (low + high)/2 [= 2]

check: F[mid] Z F[high] [4 <= 5 < 7 = true]
x := mid [= 2]

2nd Iteration

check: low+1 != high [= true]
m = (low+high)/2 [= 3]
check: F[mid] Z F[high] [5 <= 5 < 7 = true]
low := mid [= 3]

3rd Iteration

check: low+1 != high [= false]
skip

With this example, we can see the general binary search algorithm is not only simpler than the traditional variant, it is also more efficient as it terminates in one less iteration.

References

[1] Bloch, J. (2006, June 2). Research Blog: Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken. Retrieved from https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html