Comparing Variants of the Binary Search Algorithm

Shan Perera

April 22, 2017

Abstract

We compare two variations of the binary search algorithm. We examine the key differences between the traditional binary search variant and the one presented in the report: A Survey Of Binary Search. We compare and contrast the differences in the implementation of both variants and show that the traditional variant is prone to unexpected behaviour and bugs.

1 Introduction

The report will consist of the discussion of the traditional binary search algorithm, and the variant found in A Survey Of Binary Search.. In each discussion of the algorithm, we will examine the differences in the variants as well as the ease of understanding the variant itself. Following the discussion of the two algorithms is a section regarding the possible bugs and unexpected behaviour that occur in the traditional binary search algorithm.

2 Traditional Binary Search

2.1 Algorithm Discussion

The traditional binary search algorithm we plan to discuss is Jon Bentley's popular implementation found in his book *Programming Pearls*. The basic structure of the traditional binary search variant is as follows: Given a sorted array A, we are trying to find an element in the array. The binary search function will return the index of the element in the array or it will tell us that the element was not found in the array. Notice that searching an array of size n is essentially searching on an interval of [0,n-1]. This is the main property that binary search is based on. Binary search first accesses the middle element in the array. The function then checks if this element is what we're searching for. If it is what we are searching for then the operation is complete, if it is not what we are searching for then the function checks if this middle element is smaller or larger than the element being searched for, hereafter referred to as the "search element". Since the array is sorted, if the middle element is smaller than the search element, then every element before the middle element can be eliminated from the search criteria. A similar case also applies to when the middle element is larger than the search element, then every element following the middle element can be eliminated from the search criteria. We examine Bentley's implementation given below:

```
  l, r := 0, N-1
; do l ≤ r ->
    m := (l + r) / 2
  ; if a[m] < K -> l := m + 1
    [] a[m] = K -> found := true; break
    [] K < a[m] -> r := m - 1
    fi
  od
; found := false
```

Notice that the middle index is calculated using the average of the min and max variables. We will discuss this calculation in the last section and how it can lead to unexpected behaviour and bugs.

3 General Binary Search

3.1 Algorithm Discussion

The general version of binary search that we plan to discuss can be found here: A Survey Of Binary Search. This binary search approach was developed by Netty van Gasteren and Wim Feijen. It is known as the Van Gasteren-Feijen approach. It state's that binary search does not need to be applied to a sorted array, but instead it can be applied to solve a more general problem: Where *M,N* are integers, and $M < N$, let there be a relation ϕ such that M ϕ N.

We are trying to find an l such that: M $\leq$ l $<$ N $\wedge$ l ϕ (l+1); The general binary search implementation is given below:

```
{ M < N ^ M φ N}
  l, r := M, N
  { Inv: M ≤ l < r ≤ N ∧ l φ r, Bound: r - l }
  ;do l+1 ≠ r ->
    { l + 2 ≤ r }
    m := (l + r) / 2
   ;if m φ r -> l := m
    [] l φ m -> r := m
    fi
  od
  { M ≤ l < N ∧ l φ (l+1) }
```

We use textual substitution to assign M and N to the variables l and r, respectively. We have the loop invariant such that the variables *M, and N* are in the bounds of the array, with the bound function: *r - l*. The loop guard *l+1* $\neq$ *r* checks that the interval being search is not of size 1. In other words, the left edge of the array is not adjacent to the right edge of the array. When we evaluate the middle index variable m by taking the average of the left and right edges, we ensure that the condition $l < m < r$ is true and ensure that the bound *l-r* will in fact decrease. If either guards $m \phi r$ or $l \phi m$ are satisfied, then the invariant holds. To find an l such that the condition l ϕ (l+1) is satisfied, we must satisfy the following condition:

```
l φ r ∧ l < m < r  ⟹  l φ m ∨ m φ r
```

Possible relations of ϕ that would satisfy the above condition are:

```
i φ j = a[i] ≠ a[j], for some array a
i φ j = a[i] < a[j],
i φ j = a[i] ⋆ a[j] ≤ 0,
i φ j = a[i] ∨ a[j],
etc.
```

To search for a index in an array using binary search we would use the relation ϕ:

```
i φ j = a[i] ≤ K < a[j]
```

The approach then states to check whether *a[i] = K* where K is the index of the searched element. However, a problem that Van Gasteren and Feijen encountered was being unable to verify the precondition above, more specifically:

```
a[l] ≤ K < a[r]
```

Their solution was to add imaginary numbers to the beginning and end of the array. Specifically, an array with interval [0,n-1], we have the imaginary element a[-1] and a[N] such that $\leq$ for all elements x in the array, we have x $\geq$ a[-1] and x $<$ a[N]. This is equivalent to using the ϕ:

```
i φ j = (i = -1 ∨ a[i] ≤ K) ∧ (K < a[j] ∨ j = N)
```

This satisfies the following condition:

```
l φ r ∧ l < m < r  ⟹  l φ m ∨ m φ r
```

The general binary search implementation is given below:

```
{ 0 ≤ N ∧ -1 φ N }
l, r := -1, N
{ Inv: -1 ≤ l < r ≤ N ∧ l φ r, Bound: r - l }
;do l+1 ≠ r ->
  { l + 2 ≤ r }
  m := (l + r) / 2
 ;if a[m] ≤ K -> l := m
  [] K < a[m] -> r := m
  fi
od
{ -1 ≤ l < N ∧ l φ (l+1) }
;if l > -1 -> found := a[l] = K
 [] l = -1 -> found := false
 fi
```

The imaginary elements are never actually inserted into the array, they are there to verify the specifications of the general binary search program. This also allows the program to handle empty arrays.

4 Calculating the Midpoint

For very large arrays and data structures, when calculating the middle index for the binary search function using the expression $m := (l + r)/2$, it is possible to have an overflow error. Specifically, when calculating the term $l + r$. If the sum of the indexes l and r exceeds the maximum positive value of a integer (2^{31}-1) This bug had gone unnoticed for many years since binary search was first implemented. Even the binary search function implemented for the Java programming language by Sun Microsystems had this bug. The solution to this bug is the following line of code:

```
m = low + ((high - low) / 2);
```

5 Conclusion

The traditional variant of binary search is broken, the midpoint calculation is bugged. For very large arrays, which are often used in datacenters for large companies like Google, the program will throw an overflow error because the result of the addition calculation returned a number that exceeded the maximum value of an integer. A general version of binary search can be implemented using the Van Gasteren-Feijen approach. Bentley's implementation of binary search has an early exit, the *break* command which is called when the index is found, which would lead you to believe that it is the faster method, however the Van Gasteren-Feijen approach is the more favoured approach. Bentley's implementation has three comparisons in the loop, whereas the Van Gasteren-Feijen method only has two. For large values of n, the cost of the extra comparison outweighs the benefits of having an early exit. Therefore, the general version of binary search should be used with the fixed midpoint calculation.