Binary Search a Little Simpler & More Generic

Jules Jacobs

December 4, 2020

1 Introduction

The usual binary search algorithm allows us to find the location of an item x in a sorted array A, using the function $\mathsf{mid}(L,R) = \mathsf{floor}((L+R)/2)$ to cut the search range in half:

```
function binary_search(A,x){
  var L = 0
  var R = length(A) - 1
  while(L <= R){
    val m = mid(L,R)
    if(A[m] < x){
      L = m + 1
    }else if(A[m] > x){
      R = m - 1
    }else{
      return Some(m)
    }
  }
  return None
}
```

This algorithm is fine as it is, but sometimes we want to find the last position in A that has an item $\leq x$, for instance when searching for which child to descend down to in an internal node of a B-tree. Modifying the algorithm above to accomplish this may seem like a trivial task, but it turns out to be a minefield of off-by-1 errors. It is better to start from scratch and generalize.

2 A simple binary search algorithm

Suppose we have some predicate $p : Int \to Bool$, and are given indices $L < R$ such that $p(L) = false$ and $p(R) = true$. We want to find an index between L and R where p flips from $false$ to $true$:

```
// precondition: L<R && p(L)==false && p(R)==true
// requirement: if R-L > 1 then L < mid(L,R) < R
function binary_search(L,R,p){
  while(R - L > 1){
    val m = mid(L,R)
    if(p(m)) R = m
    else L = m
  }
  return (L,R)
}
// postcondition: R=L+1 && p(L)==false && p(R)==true
```

A few notes about this algorithm:

- The loop invariant $L < R \wedge p(L) = false \wedge P(R) = true$ is maintained.
- The value of $R - L$ strictly decreases on each iteration, so the algorithm terminates. In the worst case, the number of iterations is precisely $\text{ceil}((\log_2(R - L + 1)) = \min\{n \in \mathbb{N} : R - L < 2^n\}$.
- When the loop exits, we have $R - L \leq 1 \wedge L < R \implies R = L + 1$, so the returned pair (L, R) precisely indicates the point where p flips from $false$ to $true$.
- The algorithm only calls p on indices *strictly* between the initial L and R.
- We do not require that p has only one point where it flips from $false$ to $true$. If there is more than one, then binary search algorithm will find one of those points.
- Whereas the previous algorithm returned `Some(m)` or `None` depending on whether it found x, this algorithm always returns a result.

3 Searching an array

Let $A = [2, 3, 3, 3, 6, 8, 8, 9]$ be a sorted array. Given a particular number x, we can ask four different questions:

(Q1) Where is the last number $< x$?

(Q2) Where is the first number $\geq x$?

(Q3) Where is the last number $\leq x$?

(Q4) Where is the first number $> x$?

These questions can be answered by `binary_search`:

```
(Q1,Q2) = binary_search(-1, length(A), function(i){ A[i]>=x })
(Q3,Q4) = binary_search(-1, length(A), function(i){ A[i]>x })
```

The results $Q1, Q2, Q3, Q4$ answer the corresponding questions. Corner cases are handled nicely: if we ask "Where's the first number < 0 in A", then the answer is -1, and if we ask "Where's the last number > 10 in A, then the answer is $\text{length}(A)$. One can think of this as artificially putting:

$$A[-1] = -\infty \qquad\qquad A[\text{length}(A)] = +\infty$$

So that:

$$p(-1) = \text{true} \qquad\qquad p(\text{length}(A)) = \text{false}$$

Note that `binary_search(L,R,p)` only calls p on numbers strictly between L and R, so the array is never accessed out of bounds, even though we put $L = -1$ and $R = \text{length}(A)$, which are themselves out of bounds.

4 Finding a particular element

We may also ask:

- Is x in the array, and if so, where is the first x and where is the last x?
- Find the range of indices that contain x.

The second question is the easier one: it's the range $Q2 \ldots Q3$. If x does not appear in the array, then the range has $Q3 = Q2 - 1$ and the empty range indicates between which two elements x would have to be found. We answer the first question as follows:

```
function find_first(A,x){
  (_,R) = binary_search(-1, length(A), function(i){ A[i]>=x })
  if(R < length(A) && A[R] == x) return Some(R)
  else return None
}

function find_last(A,x){
  (L,_) = binary_search(-1, length(A), function(i){ A[i]>x })
  if(L > 0 && A[L] == x) return Some(L)
  else return None
}
```

Perhaps surprisingly, this binary search algorithm can be more efficient than the original binary search, provided one inlines the predicate. The original binary search loop tries to bail out early if it happens to find an element equal to x early on, but the small probability of that happening doesn't outweigh the extra test and branch on each iteration.

The `if` inside the simplified binary search can be compiled to two conditional move instructions, thus eliminating all branches in the loop. Furthemore, if the array size is known, for instance when searching B-tree nodes of 31 elements, it suffices to execute 5 iterations of the loop[1], so one can unroll:

```
L = -1
R = 32
m = mid(L,R)
if(A[m] >= x) R = m else L = m
m = mid(L,R)
if(A[m] >= x) R = m else L = m
m = mid(L,R)
if(A[m] >= x) R = m else L = m
m = mid(L,R)
if(A[m] >= x) R = m else L = m
m = mid(L,R)
if(A[m] >= x) R = m else L = m
```

For more information about optimizing binary search, read the excellent article by Paul Khuong.[2]

[1] By picking the array length to be $2^n - 1$, we can pick the middle element of the range on each iteration, and binary search will need exactly n iterations. For other array sizes there won't be a middle element on some iterations because an array of even length doesn't have a middle element. Thus, arrays of length $2^n - 1$ are optimal for binary search.

[2] http://pvk.ca/Blog/2012/07/03/binary-search-star-eliminates-star-branch-mispredictions/

5 Generalized binary search

In order for the `mid(L,R)` function to work, we need to know that $R-L>1$, so that `mid` can actually find an index i strictly between $L<i<R$. The loop test checks this condition. It therefore makes sense to combine these into the mid function:

```
mid(L,R) = if(R - L > 1) then Some(floor((L+R)/2)) else None
```

So that `mid` returns an `Option[Int]`. We can then write binary search as follows:

```
function binary_search(L,R,p){
  while(true){
    case mid(L,R){
      None    => return (L,R)
      Some(m) => if(p(m)) then R = m
                          else L = m
    }
  }
}
```

Or, in functional, recursive style:

```
function binary_search(L,R,p){
  case mid(L,R){
    None    => (L,R)
    Some(m) => if p(m) then binary_search(L,m,p)
                       else binary_search(m,R,p)
  }
}
```

By picking a different `mid` function, we get a different search:

- For forward linear search, pick
  `mid(L,R) = if(R-L > 1) then Some(L+1) else None`.
- For backward linear search, pick
  `mid(L,R) = if(R-L > 1) then Some(R-1) else None`.

5.1 Searching floats

We can also search floating point numbers, instead of integers:

- For floating point search, pick
  `mid_float(L,R) = if(R-L > eps) then Some((L+R)/2) else None`.

This allows us to bisect floating point equations, such as $x^2=2$:

```
binary_search(1.0, 2.0, function(x){ x*x >= 2 })
```

However, there is a *much* better way to do bisection on floats: instead of taking the midpoint $(L+R)/2$, we take the midpoint between the two floats L and R in the *binary representation*. That is, suppose that `f2b : Float64 -> Int64` gives you the bitwise representation and `bf2 : Int64 -> Float64` converts back, we pick the midpoint:[3]

[3] I'm assuming that `fb2` is respects ordering, that is, comparing `f2b(x) < f2b(y)` gives the same result as comparing the floats `x < y`. Depending on the bit representation of floats, one would have to shuffle the mantissa and exponent and sign bits around to ensure this.

```
function mid_float(L,R){
  case mid(f2b(L),f2b(R)){
    None => None
    Some(bits) => b2f(bits)
  }
}
```

Where `mid` is the mid function on integers. Using `mid_float`, we can determine the precise floating point number `x` at which the predicate `x*x >= 2` flips from false to true in at most 64 iterations!

5.2 Searching lattices

Instead of searching for numbers, we can even search in lattices. Suppose that we have a predicate $p : 2^S \to Bool$ on subsets of a finite set S. Given two sets $L \subseteq R$ the `mid_set(L,R)` function shall give `None` if $|R - L| = 1$ and some set M such that $L \subset M \subset R$ otherwise. Then the binary search algorithm can give us sets L, R such that $R = L \cup \{x\}$ for a single element $x \in S$, with $p(L) = false$ and $p(R) = true$. By picking M to be L plus half of the elements of $R - L$, the algorithm terminates in $O(|R - L|)$ iterations.

What else can *you* make binary search do?