IP Lecture 7: Binary Search

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Introduction

In this lecture we'll see two rather different searching problems:

- To find the integer square root of a given positive integer `y`, i.e. to find a non-negative integer `a` such that $\mathtt{a}^2 \leq \mathtt{y} < (\mathtt{a}+1)^2$.
- To search for a particular value `x` in a sorted array `a`, i.e. we'll find an index `i` such that `a(i)=x`, if such an `i` exists.

In each case we'll see two algorithms:

- The first algorithm will be a straightforward linear search.
- The second algorithm will, at each stage, keep track of a range, within which our answer lies. At each step we will (roughly) half the size of this range, leading to a logarithmic algorithm.

Integer square root

Given an integer `y`, we want to find an integer `a` such that $a^2 \le y < (a+1)^2$.

Clearly this only makes sense if $y \ge 0$; we will assume this as a precondition.

One way to solve this problem would be by a linear search, i.e. trying each value of `a` in turn:

```
// Pre: y >= 0
// Post: returns a s.t. a^2 <= y < (a+1)^2
def linearSqrt(y: Int) : Int = {
  require(y >= 0)
  var a = 0 // invariant I: a^2 <= y
  while((a+1)*(a+1) <= y) a = a+1
  // a^2 <= y < (a+1)^2
  a
}
```

Integer square root

The linear search program illustrates an important technique for coming up with invariants. Writing the postcondition as

$$\mathtt{a}^2 \le \mathtt{y} \wedge \mathtt{y} < (\mathtt{a}+1)^2$$

we took the invariant to be the first of these conjuncts:

$$I \mathrel{\widehat{=}} \mathtt{a}^2 \le \mathtt{y}.$$

We also took the guard to be the negation of the second conjunct, i.e.

$$guard \mathrel{\widehat{=}} (\mathtt{a}+1) * (\mathtt{a}+1) \le \mathtt{y}.$$

When the loop terminates, we will have $I \wedge \neg guard$; but this implies the postcondition by the way we chose I and $guard$.

But this algorithm takes $\Theta(\sqrt{\mathtt{y}})$ steps. We can do better.

Binary search

In the previous program, at each point the desired result (the integer square root) was somewhere in the infinite range $[\mathtt{a}..\infty)$.

In our next program, at each point the desired result will be somewhere in a finite range $[\mathtt{a}..\mathtt{b})$. That is, we will use the invariant:

$$\mathtt{a}^2 \leq \mathtt{y} < \mathtt{b}^2 \land 0 \leq \mathtt{a} < \mathtt{b}.$$

At each step, we will (roughly) half the size of this range, by picking a value `m` roughly half way between `a` and `b`, and subsequently searching either in the range $[\mathtt{a}..\mathtt{m})$ or the range $[\mathtt{m}..\mathtt{b})$.

Binary search

```
require(y >= 0)
// Invariant I: a^2 <= y < b^2 and 0 <= a < b
var a = 0; var b = y+1
while(a+1 < b){
  val m = (a+b)/2 // a < m < b
  if(m*m <= y) a = m else b = m
}
// a^2 <= y < (a+1)^2
```

- Check the assertion `a < m < b`.
- Check the conditions for correctness.
- The loop performs about $\log_2 y$ iterations.
- Could we have written the guard as `a < b`?

Searching in an array

Suppose we have an array `a[0..N)` (not necessarily in order) and we want to find whether a particular value `x` occurs in `a`, and if so the index at which it occurs. We can do so using a straightforward linear search, with invariant

$$(\forall j \in [0..\mathtt{i}) \bullet \mathtt{a}(j) \neq \mathtt{x}) \wedge 0 \leq \mathtt{i} \leq \mathtt{N}$$

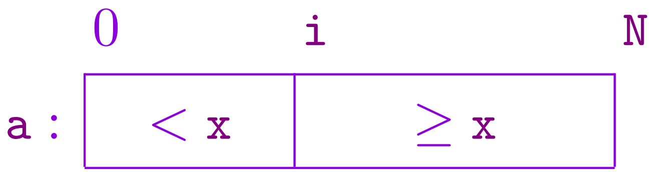
```
var i = 0
// Invariant: for all j in [0..i), a(j) != x and 0 <= i <= N
while(i < N && a(i) != x) i = i+1
```

Note that this finds the first occurrence of `x`. Also note that this sets `i` to `N` if `x` does not appear in the array.

Searching in an ordered array

Now suppose we have an ordered array `a[0..N)`, i.e. such that $\mathtt{a}(i) \le \mathtt{a}(j)$ whenever $0 \le \mathtt{i} \le \mathtt{j} < \mathtt{N}$. Again we want to find whether a particular value `x` occurs in `a`, and if so the index at which it occurs.

In fact, the program will find an index `i` in `[0..N]` such that `a[0..i) < x` $\le$ `a[i..N)`, which we can picture as follows:



pre: `a` is sorted

post: **returns** `i` **s.t.** $\mathtt{i} \in [0..\mathtt{N}] \wedge \mathtt{a}[0..\mathtt{i}) < \mathtt{x} \le \mathtt{a}[\mathtt{i}..\mathtt{N})$

Searching in an ordered array

Once we have found `i` such that `a[0..i) < x` $\leq$ `a[i..N)`, we can then use a piece of code such as

```
if(i < N && a(i) == x) println("Found at position "+i)
else println("Not found")
```

Alternatively, suppose we want to insert the value `x` into the array, while keeping it sorted, we will insert it at position `i` (this is the basis for the Insertion Sort algorithm).

The invariant

By comparison with the previous example, we might consider using an invariant such as `a(i)` $\leq$ `x < a(j)`. But how can we find `i` and `j` to establish this?

We could invent ficticious entries `a(-1) =` $-\infty$ and `a(N) =` $+\infty$, and carefully avoid accessing those entries. But ....

The invariant

Things will be easier if we use the invariant

$$\mathtt{a}[0..\mathtt{i}) < \mathtt{x} \leq \mathtt{a}[\mathtt{j}..\mathtt{N}) \wedge 0 \leq \mathtt{i} \leq \mathtt{j} \leq \mathtt{N}$$

We can picture this as follows:

$$\begin{array}{r|c|c|c|}
 & 0 \qquad\quad \mathtt{i} & \qquad\qquad \mathtt{j} & \qquad\qquad\quad \mathtt{N} \\
\hline
\mathtt{a}: & < \mathtt{x} & ? & \geq \mathtt{x} \\
\hline
\end{array}$$

In effect, this says that we have narrowed the search range down to `[i..j]`. At each iteration, we will pick a value `m` roughly in the middle, and then continue searching in either the range `[i..m]` or the range `[m+1..j]`.

Searching in an ordered array

```
// invariant I: a[0..i) < x <= a[j..N) && 0 <= i <= j <= N
var i = 0; var j = N
while(i < j){
  val m = (i+j)/2 // i <= m < j
  if(a(m) < x) i = m+1 else j = m
}
// I && i = j, so a[0..i) < x <= a[i..N)
```

- Check the assertion `i <= m < j`.
- Check the conditions for correctness.
- Why do we set `i=m+1` in the "then" case? Could we set `j=m-1` in the "else" case?
- Could we jump out if we find `a(m) == x`?
- The loop performs about $\log_2$ `N` iterations.

Binary search

From Wikipedia:[a]

> When Jon Bentley assigned it as a problem in a course for professional programmers, he found that an astounding ninety percent failed to code a binary search correctly after several hours of working on it[b] and another study shows that accurate code for it is only found in five out of twenty textbooks[c]. Furthermore, Bentley's own implementation of binary search, published in his 1986 book Programming Pearls, contains an error that remained undetected for over twenty years[d].

---

[a] `http://en.wikipedia.org/wiki/Binary_search_algorithm`.

[b] Jon Bentley, Programming Pearls, Addison-Wesley, 1986.

[c] Richard E. Pattis, Textbook errors in binary searching, SIGCSE Bulletin, 20, 1988, 190–194.

[d] Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken, Google Research Blog.

A test suite for binary search

Many programs go wrong on boundary cases (see lecture on testing); the following test suite concentrates on such boundary cases. It also checks for correctness when the value searched for appears never, once, or more than once.

```
class BinarySearchTests extends FunSuite{
  val a = Array(2,4,6,6,8,10); val b = Array(2,2,4,6,6,8,10,10)
  test("1. before first"){ assert(search(a,0) === 0) }
  test("2. matches singleton at start"){ assert(search(a,2) === 0) }
  test("3. matches repeated value at start"){ assert(search(b,2) === 0) }
  test("4. matches singleton in middle"){ assert(search(a,4) === 1) }
  test("5. matches repeated value in middle"){ assert(search(a,6) === 2) }
  test("6. missing item"){ assert(search(a,7) === 4) }
  test("7. matches singleton at end"){ assert(search(a,10) === 5) }
  test("8. matches repeated value at end"){ assert(search(b,10) === 6) }
  test("9. greater than last"){ assert(search(a,11) === 6) }
  test("10. empty array"){ assert(search(Array(), 5) === 0) }
}
```

Equivalence class partition testing

We can partition the set of all searches over non-empty arrays into a finite number of equivalence classes, depending upon (a) the number of times the searched value x occurs; (b) how x compares with the minimum and maximum elements of the arrays. The following table shows which test considers which partition; partitions that cannot occur are marked "–".

#	$x < min$	$x = min$	$min < x < max$	$x = max$	$max < x$
0	1	–	6	–	9
1	–	2	4	7	–
> 1	–	3	5	8	–

If the program works correctly for one test in a partition, it is likely to work for all inputs in that partition.

Summary

- Binary search for integer square root;
- Binary search in an array;
- Partition testing.
- Next time: Quicksort.