Counting the Steps of an Iterative Binary Search Algorithm

Stefan Bruda

CS 317

The algorithm analyzed here goes like this:

```
algorithm BinSearch(x, S, l, h):
    i ← l
    j ← h
    while i ≤ j do
        m ← (i + j)/2
        if S_m = x then return m
        else if S_m > x then j ← m − 1
        else i ← m + 1
    return −1
```

The algorithm inspects all the values in S between l and h. Therefore $n = h - l$ makes sense as input size.

We want now to count the number of iterations performed by the main (and only) loop. For this purpose we first note that the loop condition ($i \leq j$) must be true for as long as the algorithms operates that is, $j - i \geq 0$.

We use j_k and i_k to denote the values of i and j at the beginning of iteration k of the while loop, with the first iteration being iteration 0[1]. Clearly, $j_0 = h$ and $i_0 = l$ and so $j_0 - i_0 = n$. Let's see how $j_{k+1} - i_{k+1}$ can be expressed as a function of i_k and j_k (and hopefully as a function of $j_k - i_k$).

We have $m = (i_k + j_k)/2$. Depending on which branch is taken in the conditional (ignoring the one that terminates the program since we perform a worst-case analysis), either:

1. $j_{k+1} = (i_k + j_k)/2 - 1$, $i_{k+1} = i_k$, and so $j_{k+1} - i_{k+1} = (i_k + j_k)/2 - 1 - i_k = (j_k - i_k)/2 - 1$; or
2. $i_{k+1} = (i_k + j_k)/2 + 1$, $j_{k+1} = j_k$, and so $j_{k+1} - i_{k+1} = j_k - (i_k + j_k)/2 - 1 = (j_k - i_k)/2 - 1$.

Luckily we end up with the same expression: $j_{k+1} - i_{k+1} = (j_k - i_k)/2 - 1$[2].

Our goal however is to find a relation between k and n, which will hopefully provide an upper bound for k. We have not accomplished that just yet, so we need to continue. We will do so by

[1] Actually, there is nothing wrong with starting at iteration 1 as usual, and very little changes below if we do that. It is just that the resulting expressions turn up nicer when starting at 0...

[2] Recall that the running time of the recursive binary search algorithm is given by the recurrence relation $T(n) = T(n/2) + 1, T(1) = 1$. This is eerily similar with what we go there (except that we start from n rather than 1 and so the +1 transforms into a −1. This is actually no accident, since the recursive and non-recursive algorithms perform the same task in essentially the same way, so it makes plenty of sense that they have roughly the same properties.

seeing how $j_k - i_k$ looks like for a few iterations, in the hope of finding a pattern:

$$\begin{array}{lcll} j_0 - i_0 & = & h - l & = & n \\ j_1 - i_1 & = & (j_0 - i_0)/2 - 1 & = & n/2 - 1 \\ j_2 - i_2 & = & (j_1 - i_1)/2 - 1 & = & n/4 - 3/2 \\ j_3 - i_3 & = & (j_2 - i_2)/2 - 1 & = & n/8 - 7/4 \\ j_4 - i_4 & = & (j_3 - i_3)/2 - 1 & = & n/16 - 15/8 \end{array}$$

The case $k = 0$ feels ever so slightly different from the others, but even so at this point I feel confident that $j_k - i_k = n/2^k - (2^k - 1)/2^{k-1}$. This is just a guess though, and as a matter of principle all guesses need to be verified. I will do so in this particular case by induction over k:

- Base case: $j_0 - i_0 = n/2^0 - (2^0 - 1)/2^{-1} = n/1 - 0/2^{-1} = n$, as desired.
- Induction step: We assume by induction hypothesis that $j_k - i_k = n/2^k - (2^k - 1)/2^{k-1}$. Then $j_{k+1} - i_{k+1} = (j_k - i_k)/2 - 1 = (n/2^k - (2^k - 1)/2^{k-1})/2 - 1 = n/2^{k+1} - (2^k - 1)/2^k - 1 = n/2^{k+1} - (2^k - 1)/2^k - 1 = n/2^{k+1} - (2^k - 1 + 2^k)/2^k = n/2^{k+1} - (2^{k+1} - 1)/2^k$, again as desired.

Now, as I already said, an iteration k of the while loop exists iff $j_k - i_k \geq 0$ that is, $n/2^k - (2^k - 1)/2^{k-1} \geq 0$. This in turn is equivalent to $n/2^k \geq (2^k - 1)/2^{k-1}$ that is, $n \geq 2(2^k - 1)$ that is, $(n + 2)/2 \geq 2^k$. It follows that $k \leq \log_2(n + 2)/2 = \Theta(\log n)$.

The obvious conclusion is that we cannot have more than $\Theta(\log n)$ iterations and so the iterative binary search algorithms has a time complexity of $O(\log n)$. It is however a mistake to conclude that the algorithm runs in $\Theta(\log n)$ time, since there is nothing in the above argument that limits k from below. It turns out that $O(\log n)$ is actually tight (meaning that the running time of the algorithm is in $\Theta(\log n)$), but that requires some additional (though fairly simple) considerations; these are left as an exercise for the interested reader).