This handout gives pseudocode for the $\Theta(\log n)$ binary search algorithm. As an exercise, think about how you would implement this algorithm \textit{without} making any recursive calls.

\textbf{Input}: a sorted array $A[p \ldots r]$, defined for indices between $p$ and $r$. Also, a number $x$.

\textbf{Returns}: the index of $x$ in $A$ if it is present; otherwise, the special value \textit{notFound}. If $x$ appears several times in $A$, one of its indices is returned.
\textsc{Bsearch}(x, A, p, r) \\
\hspace{1em} \textbf{if} \ p = r \hspace{1em} \triangleright \text{base case} \\
\hspace{1em} \hspace{1em} \hspace{1em} \textbf{if} \ A[p] = x \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{return} \ p \\
\hspace{1em} \hspace{1em} \hspace{1em} \textbf{else} \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{return} \ \textit{notFound} \\
\textbf{else} \\
\hspace{1em} \text{mid} \leftarrow \lceil (p + r)/2 \rceil \\
\hspace{1em} \textbf{if} \ A[\text{mid}] > x \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{return} \ \textsc{Bsearch}(x, A, p, \text{mid} - 1) \\
\hspace{1em} \textbf{else} \\
\hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \textbf{return} \ \textsc{Bsearch}(x, A, \text{mid}, r)
1 Correctness

To see correctness, start with an invariant: at the beginning of each call to the Bsearch function, if the value \( x \) occurs in \( A \), it lies in \( A[p..r] \). The invariant is an important property of the algorithm that holds at all times while it is running. As we’ll see, the invariant is useful in understanding why the algorithm is correct.

We will prove the following claim: suppose the invariant holds at the beginning of the call to Bsearch. Then the algorithm returns the index of \( x \) in \( A \) if \( x \in A \), or notFound if \( x \notin A \). Since the invariant trivially holds at the start of the top-level call to Bsearch, this claim implies the correctness of the algorithm.

**Pf:** The proof is by induction on \( n = r - p + 1 \), the size of \( A \). In the base case, if \( n = 1 \) (equivalently, \( p = r \)), \( A \) consists of the single element \( A[p] \), and the base case code clearly returns the correct answer given the invariant.

In the general case, suppose the invariant holds on entry to Bsearch; hence, if \( x \) is in \( A \) at all, it is in \( A[p..r] \). We first argue that, regardless of which of the two recursive calls the algorithm makes, it does so on a strictly smaller subarray. Indeed, if \( r = p + 1 \), then \( \text{mid} \) is \( r \), and so the recursive calls would be on subarrays \( A[p] \) and \( A[r] \), respectively. If \( r > p + 1 \), then \( \text{mid} \) lies strictly between \( p \) and \( r \), and so the two calls again use smaller subarrays.

We now argue that the code preserves the invariant in whichever recursive call it makes. There are two cases.

- If \( x < A[\text{mid}] \), then if it occurs in \( A \) at all, it must occur at a position prior to \( \text{mid} \). Hence, the invariant is preserved if we recur on \( A[p..(\text{mid} - 1)] \).

- If \( x \geq A[\text{mid}] \), then if it occurs in \( A \) at all, it must occur at position \( \text{mid} \) or later. Hence, the invariant is preserved if we recur on \( A[\text{mid}..r] \).

In either case, the recursive call is made on a strictly smaller subarray of the input while satisfying the invariant, so by our inductive hypothesis (i.e. the original claim), that call returns the correct answer. QED

2 Complexity

We can use recursion tree analysis to determine the running time of Bsearch. Assume for simplicity that the input array is a power of two in length. Each call to Bsearch on an array of size \( > 1 \) makes one recursive call and does constant additional work to determine which call to make. Hence, a recurrence for the running time \( T(n) \) is

\[
T(n) = \begin{cases} 
  c_0 & \text{if } n = 1 \\
  T(n/2) + c & \text{if } n > 1.
\end{cases}
\]

See the class notes for a recursion-tree solution to this recurrence. The running time turns out to be \( \Theta(\log n) \), which is much faster than a naive linear search through \( A \).