1 Time Complexity of Binary Search in the Worst Case

As linear search, the worst case for binary search is when the value x is not in the set L as then the algorithm has to perform the maximum possible number of recursive calls, and hence the maximum possible number of operations.

```
**Algorithm** BinarySearch($L, x, first, last$
**Input:** Array $L[first, last]$ and value $x$.
**Output:** $-1$ if $x \notin L$, or $i$, $0 \leq i < n$ if $L[i] = x$.

if $first > last$ then return $-1$
else {
    $middle \leftarrow \lfloor \frac{first+last}{2} \rfloor$
    if $L[middle] = x$ then return $middle$
    else if $L[middle] < x$ then return BinarySearch($L, x, middle+1, last$)
    else return BinarySearch($L, x, first, middle-1$)
}
```

Perhaps the easiest way of computing the time complexity of a recursive algorithm, like the one above, is by expressing first the complexity as a recurrence equation and then solving that equation. A recurrence equation has 2 parts, a case base an a recursive part, matching the two parts of a recursive algorithm.

Let $f(n)$ denote the number of operations that algorithm BinarySearch needs to perform when the size of the input is n. Then, $f(0) = c'$, where c' is a constant, as when the size of the array is zero, the algorithm just performs a constant number of operations (a comparison and a return operation). When the size of the array is larger than zero, then the algorithm performs a constant number c of operations to find the element in the middle of L, compare that element with x and decide whether the algorithm must be invoked recursively on the left half or on the right half of the array. We will make the simplifying assumption that both halves of the array have the same size, $(n-1)/2$. Hence, the total number of operations that the algorithm performs when $n > 0$ is $f(n) = c + f((n-1)/2)$. The recurrence equation giving the time complexity of linear search, then, is

$$\begin{aligned} f(0) &= c' \\ f(n) &= c + f(\frac{n-1}{2}) \text{ if } n > 0 \end{aligned}$$

We solve this equation by repeated substitution:

$$\begin{aligned}
f(n) &= c + f(\frac{n-1}{2}) && (1)\\
f(\frac{n-1}{2}) &= c + f(\frac{\frac{n-1}{2}-1}{2}) = c + f(\frac{n-1-2}{2^2}) = c + f(\frac{n-2^0-2^1}{2^2})\\
f(\frac{n-2^0-2^1}{2^2}) &= c + f(\frac{n-2^0-2^1-2^2}{2^3})\\
&\vdots\\
f(\frac{n-2^0-2^1-\cdots 2^j}{2^{j+1}}) &= c + f(\frac{n-2^0-2^1-\cdots 2^{j+1}}{2^{j+2}})
\end{aligned}$$

As you can size the value of the argument on the terms in the left hand size decreases with each new equation. We stop this process when the value of the argument is equal to zero, i.e., when

$$\frac{n-2^0-2^1-\cdots 2^{j+1}}{2^{j+2}} = 0$$

Simplifying, we get that

$$\begin{aligned}
f(n) &= c + c + c + \cdots + c + f(\frac{n-2^0-2^1-\cdots 2^{j+1}}{2^{j+2}})\\
&= c + c + c + \cdots + c + f(0)\\
&= c + c + c + \cdots + c + c' && (2)
\end{aligned}$$

The above set of equations (1), has $j+2$ equations, hence the number of c terms in (2) is $j+2$ and so $f(n) = (j+2)c + c'$. To determine the value for j we use the fact that

$$\frac{n-2^0-2^1-\cdots 2^{j+1}}{2^{j+2}} = 0$$

Simplifying, we get

$$\begin{aligned}
n &= 2^0 + 2^1 + \cdots + 2^{j+1}\\
&= \sum_{i=0}^{j+1} 2^i = 2^{j+2} - 1
\end{aligned}$$

Therefore, taking logarithms on both sides of this last equality, we get $j+2 = \log_2(n+1)$. Then, $f(n) = c\log_2(n+1) + c'$. Ignoring constant terms, we finally conclude that $f(n) = O(\log n)$.