10. Data Structures and Algorithms :: Time Complexity of Binary Search

Let's analyze the time complexity of the recursive binary search algorithm (written as pseudocode):

```
Begin recursiveBinarySearch(In: List<T> list; In: T key; In: low; In: high) → Int
  If low > high Then
    Return NOT_FOUND
  End If
  middle := (low + high) / 2
  If key = list_middle Then
    Return middle
  ElseIf key < list_middle Then
    Return recursiveBinarySearch(list, key, low, middle - 1)
  Else
    Return recursiveBinarySearch(list, key, middle + 1, high)
  End If
End recursiveBinarySearch
```

10. Analysis of Algorithms :: Analysis of Binary Search (continued)

Binary search is a bit more complicated to analyze than linear search, so in this discussion we will gloss over some of the more minute details rather than providing a rigorous proof.

Let n be the size of the list being searched; without loss of generality (it makes the proof simpler) we shall assume that n is a power of 2, i.e., $n = 2^p$ for $p \geq 0$.

The key operation is the comparison of *key* to $list_{middle}$ and although we actually make two comparisons during each recursive call, we will treat this as one comparison (we will obtain the same time complexity in either case).

Let $c(n)$ be the maximum number of comparisons in the worst case. But what is the worst case? As with *linearSearch*, the worst case (the maximum number of comparisons) occurs when *key* is not in *list*.

If *key* is not in *list* then we will make recursive calls—reducing the size of the list by exactly half each time since we assume n is a power of 2—until the base case of an empty list is reached. This means that $c(n)$ will be proportional to the number of recursive calls, so all we need to do to find $c(n)$ is determine the number of recursive calls.

10. Analysis of Algorithms :: Analysis of Binary Search (continued)

Let's number the calls to *recursiveBinarySearch* 1, 2, 3, ..., k. A table will be helpful:

Call	**List Size**	**Comparisons**
1	n	1
2	n / 2	1
3	n / 4	1
...	...	...
k - 1	n / 2^{k-2}	1
k	0	0

Note that on call k, the base case, the list will be empty so no comparison will be made. Therefore, the number of comparisons $c(n) = k - 1$ and our question becomes, what is $k - 1$?

Since n is a power of 2 (by our assumption), on call $k - 1$ the list size will be 1. Thus:

$$n / 2^{k-2} = 1$$

Solving for k:

$$n = 2^{k-2}$$
$$lg\ n = lg\ 2^{k-2}$$
$$k - 1 = (lg\ n) + 1 = c(n).$$

$c(n)$ can easily be shown to be $O(lg\ n)$ so the worst case asymptotic time complexity of binary search is $O(lg\ n)$.

10. Analysis of Algorithms :: Analysis of Binary Search (continued)

How does that compare to linear search?