Binary Search

Explanation

This is a classic algorithm to efficiently search for a target value in a sorted list of numbers. The condition though, is that the input list of numbers should be sorted before the algorithm executes. To explain binary search, we follow an example:

$$I = \{1, 3, 5, 17, 19, 23, 41\}$$

Here, I is a list of input numbers. Notice that numbers in I are **sorted** in ascending order. Say we have a target number t that we want to search for:

$$t = 41$$

Step 1: We start by comparing the middle element with our target element t. If the length of I is odd, we split the array at:

$$i = ceil(I.length/2)$$

where,

$ceil(x) = smallest\ integer \geq x$

$I.length = length\ of\ array\ I$

If I is even, we split the array at index position i as:

$$i = (I.length/2)$$

Assuming the array indices start at 1, in this situation, $i = ceil\left(\frac{7}{2}\right) = 4$

$I[i] = I[4] = 17 < t$

So this means, our target is in the half that comes after i. So we next check in $I1 = \{19, 23, 41\}$

Step 2: In the new array $I1$, we now need to look at the middle element again to see if we have found our target element. In this case $I1$ has an odd number of elements. So,

$i = ceil\left(\frac{3}{2}\right) = 2$

$I1[i] = I1[2] = 23 < t$

t is still greater than the middle element! We need to keep looking.

Step 3: Since in the previous step, t was greater than the middle element, we now have a new array to search in: $I2 = \{41\}$

So this is a single element array. Which means, if this element is not equal to our required target t, we exit the algorithm and return a -1/false/something that tells that the number searched for, is not in I. In our case,

$I2[0] = 41 = t$

We terminate the algorithm with a true/index position 7/something that tells the required target was found.

Complexity Analysis

At each step of the algorithm, we search in an array no more in size than $n/2$ where n is the total size of the input array. This means, our search space is getting halved each time we split the input array into two parts.

Let us assume that after k steps into the algorithm, our algorithm terminates with no result found. To compute the complexity, we need to find what the limits on k are. After first step, we have a maximum of $n/2$ elements to search from. After the second step, we have $n/4$, after the third step $n/8$ and so on. We can generalize this to the following statement:

"After k steps into a binary search algorithm, we have, no more than $n/2^k$ numbers in the array to search from"

The above, is equivalent to stating:

$$n - \left\lfloor \frac{n}{2} \right\rfloor < n/2$$

where,

$\left\lfloor \frac{n}{2} \right\rfloor$ is just a more mathematical way of writing $ceil(n/2)$

The above inequality can be proven by mathematical induction and a rigorous proof is left to the reader as an exercise (**hint:** The $ceil(x)$ has a property, that for any integer a, $\lfloor x + a \rfloor = \lfloor x \rfloor + a$)

Suppose we went on dividing the input array to such an extent, that $\frac{n}{2^k} < 1$. This basically means that we were not able to find the target element that we set out to search for. However, the step before we took this eye-opening kth step, we did have a valid array to search in. This means, $\frac{n}{2^{k-1}} \geq 1$. Putting these inequalities together,

$$\frac{n}{2^k} < 1 \leq \frac{n}{2^{k-1}}$$

Inverting the terms in the inequality, we have

$$\frac{2^k}{n} > 1 \geq \frac{2^{k-1}}{n}$$

which gives us

$$2^k > n \geq 2^{k-1}$$

Taking base 2 logarithms throughout the inequality,

$$k > \log_2 n \geq k - 1$$

Adding 1 throughout the inequality gives us

$$k + 1 > \log_2 n + 1 \geq k$$

Now, using the left half of the last-but-one inequality, and the right half of the last inequality, we have

$$\boldsymbol{\log_2 n < k \leq \log_2 n + 1}$$

And so, the time complexity of binary search, is bounded by $\log_2 n + 1$. So it is $\boldsymbol{O(\log_2 n)}$ in time.