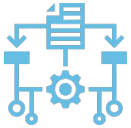
Algorithms: Binary Search Part-1

Relevel
by Unacademy

Table Of Contents



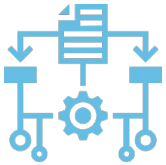
Searching Algorithms

Linear Search

Binary Search

Binary Search Questions

Searching Algorithms



In computer science, the algorithms which are used to find or search an element(s) in a given set of data are defined as searching algorithms. Searching algorithms are applied depending on the nature of the dataset. Here in this module, we would be focusing on two prime searching algorithms which are

- Linear search
- Binary search

The third algorithm which is not so widely used is the Ternary Search. We'll be discussing that too.

Linear Search

Suppose we've given an array of elements. Let's name the array as input array.
Input Array -

Now, we have been asked to check whether an element 3 is present in the input array or not. If it is present, return the index of the element in the input array.
[Index are named with red colors]
How we will approach this problem!?

Linear Search

The brute force way that everyone can think of is - Let's start from the beginning of the input array and traverse it. We check whether the current element is the same as the given element to be searched as we traverse ahead. If it fulfills our condition, we simply return the element's true and index. True denotes that the element is present. Otherwise, if we reach the end of the array(one ahead of the last index, this means we have not found the desired element. In this case, we will return false and index as -1.

This approach which we used above, is called Linear Search. Simple and very intuitive.

Linear Search

Now a few points that we can note here. We can also start the traversal from the end of the input array. Linear search can be applied to unsorted arrays or even to sorted arrays. Pseudo Code

1. We make a variable start and initialize it with 0.
2. Traverse the array by incrementing the start variable using a loop. We simultaneously check whether the element at the start index is the same as the given element to be searched. If the element is found, we simply return the start index.
3. Otherwise, we'll come out of the for loop and return -1. -1 indicates the element is not present in the given array.

Linear Search

Code Link : https://ideone.com/mLxV8T
Here we go through all the elements in an array and compare each element with the key. If we find a match, we return the index of the element. In our case, the variable i keeps track of where we are in the array, and if we find a match, we return the current value for i.

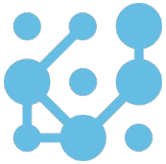
If the element doesn't exist in our list, the linear search function won't return any i value from the loop. We just return -1 after the loop to show that the function didn't find the desired element.

Global Linear Search

In the previous implementation, we return a value after we come across the first element we are looking for(key). But what if we want to find the indices of all the occurrences of a given element.
That's where global linear search comes in. It is a variant of linear search where we are looking for multiple occurrences of a given element.

Code Link : https://ideone.com/lqlEnv

Complexity of Linear Search



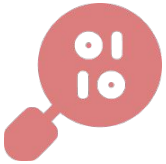
Linear Search is a classic example of a brute-force algorithm. This means that the algorithm doesn't use any algorithmic logic to optimize or somehow reduce the range of elements it searches for the key. Other search algorithms aim to do this more efficiently by preprocessing the list/array, for example, sorting it. Since we are traversing the array once, either from the beginning of the array or the end of the array. The upper bound complexity is O(n), n is the size of the given array.

Complexity of Linear Search

The time complexity of Linear Search is O(n), where n is the number of elements in the list we're searching. This is because we always consider the worst-case while calculating the time complexity. In the case of Linear Search (as with most search algorithms), the worst-case occurs when the element doesn't exist in the list. In this situation, we'd need to go through all the n elements to determine that the element isn't there.

Space Complexity - O(1)

Binary Search



Binary Search is a very simple, intuitive, yet efficient searching algorithm. The only caveat is that it works only on monotonically arranged sequences. Yes, y'all read it right!. As per the prevalent word in the market, Binary Search works on sorted sequences. The better and the more mathematically correct statement would be that "Binary Search is applicable on monotonic sequences."

Binary Search

What is monotonicity?

The monotonicity of a function tells if the function is increasing or decreasing.

What is a monotonic function?

A monotonic function increases in its entire domain or decreases in its entire domain.

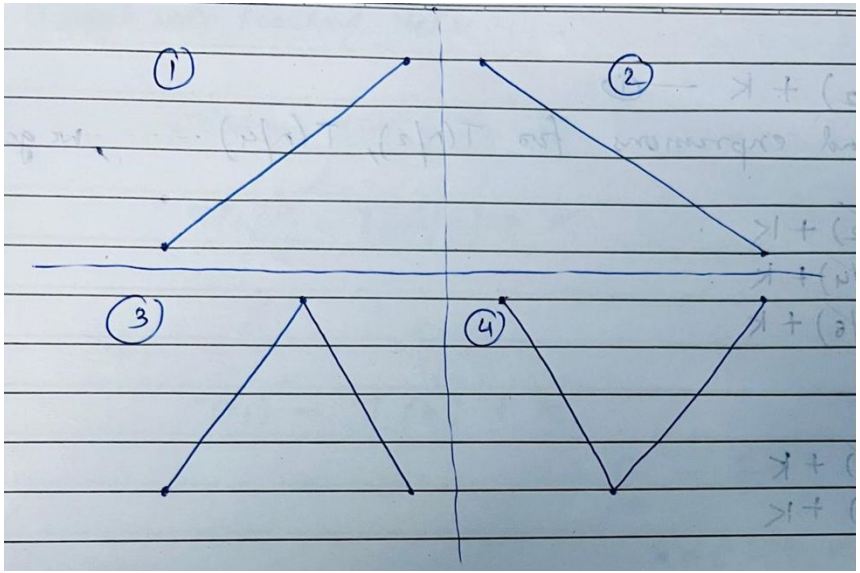
What is a non-monotonic function?

A non-monotonic function is increasing and decreasing in different segments of the domain.

Binary Search

1) Strictly Increasing Monotonic Sequence.
2) Strictly Decreasing Monotonic Sequence.
3) Non Monotonic Sequence
4) Non Monotonic Sequence.

Binary Search can only be applied to monotonic sequences.



Binary Search

Binary Search is a divide-and-conquer algorithm that divides the array roughly in half every time. Further on, we continue the algorithm in the favorable search space, which is half of the original.
In this way, we are discarding half of the search space at every step.
In other words, we divide the problem into simpler problems until it becomes simple enough to solve them directly.

Binary Search

Let's assume we have a sorted array (in ascending order) and take a look at the steps of binary search:

1. Find the middle element of the given array.
2. Compare the middle element with the value we are looking for (called key).
    - Search in the left half if the key is less than the middle element.
    - Search in the right half if the key is more than the middle element.
    - If the key is equal to the middle element, return the index of the middle element.
1. Continue with steps 1, 2 until we are left with a single element.
2. If the key is still not found, return -1.

Binary Search

To understand this better, let's look at an example of why we can simply discard half of the current search range each time we check an element:

Initial array: 1 2 5 7 13 15 16 18 24 28 29
Looking for 18

Start at the middle: 1 2 5 7 13 **15** 16 18 24 28 29
Is 18 larger/smaller than or equal to 15?
Larger, so 18 must be in the "right" half of the array.

We can continue our search in that half: 16 18 24 28 29

Binary Search

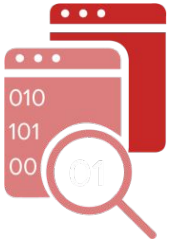
Similarly to this first split, we can keep dividing the array until we either find the element, or end up with only one candidate for the key.

Current array: 16 18 24 28 29
Is 18 larger/smaller than, or equal to 24?
Smaller, so 18 must be in the "left" half of the array.

Current array: 16 18
Which one we choose depends on how the code was written (since there's an even number of elements), but let's assume we chose 16.
Is 18 larger/smaller than, or equal to 16?
Larger, so it must be in the "right" half of the array.
Notice that the left half doesn't exist, if we had to go left we would conclude that the element we were looking for wasn't in the array.

Current array: 18
Is 18 larger/smaller than, or equal to 18?
Equal, we have found 18.

Binary Search



In this case, we ended up with only one possible candidate for the key, and it turned out to match the element we were looking for.
As you can see in the example, it took us relatively few comparisons to find the needed element. Namely, we only needed to make four comparisons to find the element in an array of 11 elements. We'll take a closer look at the efficiency of Binary Search later, but it should already be clear that it overpowers simple searching algorithms like Linear Search.

Code Link for Iterative Implementation of Binary Search :
https://ideone.com/HkVSiy

Code Link for Recursive Implementation of Binary Search :
https://ideone.com/ZczkNY

Complexity of Binary Search

Analyzing the code to obtain the recurrence relation for the time complexity of the algorithm.., we observe that at every step, we are doing two things -

a. Reducing the search space to half.

b. Computing constants like calculating the middle, assigning start or end(depending on the condition triggered)

This leads us to the following expression -

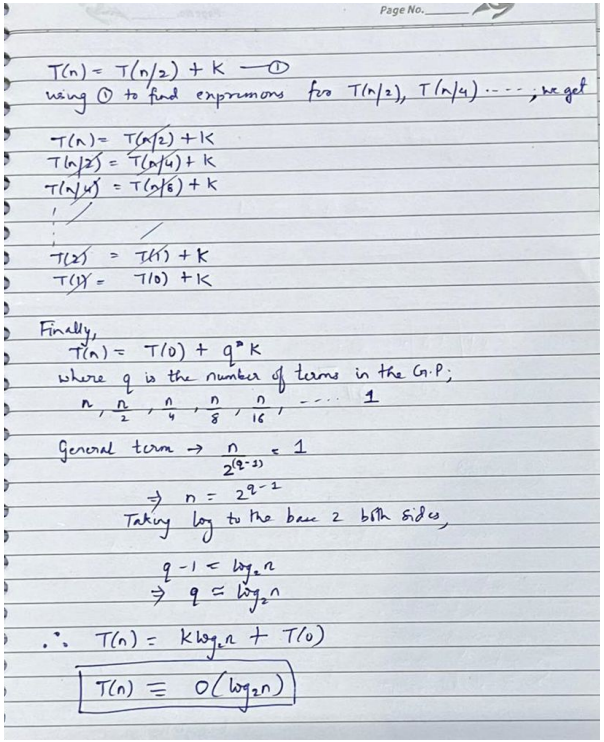
$T(n) = T(n/2) + K$

T(n) => Mathematical Function denoting Time complexity of Binary Search Algorithm.

K => Time taken in constant computing work like calculating the middle, assigning start or end.

Solving the equation, - next page

Complexity of Binary Search



$T(n) = T(n/2) + k$ — ①

using ① to find expressions for $T(n/2)$, $T(n/4)$ ····, we get

$T(n) = T(n/2) + k$

$T(n/2) = T(n/4) + k$

$T(n/4) = T(n/8) + k$

$T(2) = T(1) + k$

$T(1) = T(0) + k$

Finally,

$T(n) = T(0) + q^* k$

where q is the number of terms in the G.P;

$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \cdots 1$

General term $\rightarrow \frac{n}{2^{(q-1)}} = 1$

$\Rightarrow n = 2^{q-1}$

Taking log to the base 2 both sides,

$q - 1 = \log_2 n$

$\Rightarrow q = \log_2 n$

$\therefore T(n) = k \log_2 n + T(0)$

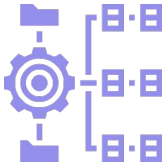
$T(n) = O(\log_2 n)$

Complexity of Binary Search

The time complexity of the Binary Search is O(log2n), where n is the number of elements in the array. This is far better than the Linear Search, which is of time complexity O(n). Like many other search algorithms, Binary Search is an in-place algorithm. That means it works directly on the original array without making any copies.

However, we have to keep in mind that Binary Search only works on sorted arrays. If using an efficient sorting algorithm, the sorting step itself has a complexity of O(nlogn). This means that in most cases, if the array is small, or if we need to search it only once, a brute-force (e.g. Linear Search) algorithm might be better.

Complexity of Binary Search



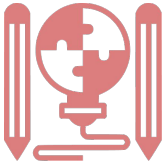
Space Complexity for Binary Search - O(1). It is an in-place algorithm. The algo doesn’t utilize any extra space.

How to approach Questions for Binary Search?

We try to simulate the given monotonic sequence as TTTTTTFFFF... or FFFFFTTTTT... And then, according to the desired condition, we are either searching for the first “F” or the first “T”.

For example, in the above question, we simulated the expression as FFFFTTTTT. And we were looking for the first “T”. The chief protagonist in this design is the ok(). We code the ok() so that our expression is maintained, and hence we navigate in the correct direction of search.

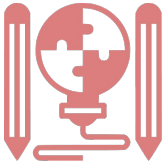
Practice Problem



Practice Problem - https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/

Code - https://ideone.com/Y3XzQd

Practice Problem



Explanation - Here, we are looking to find the first and last position of the element in the sorted array. Hence we are applying binary search to find the leftmost occurrence of the element and similarly applying binary search to find the rightmost occurrence of the element.

In the code, for finding the rightmost occurrence, why are we initializing hi = a.length, whereas in the modified template three, we discussed that it should be initialized to a “possibly correct” value? By initializing hi = a.length, we take care of the case when the element is not present in the array! We could have handled this case separately, but it makes the code more compact.

For finding out the leftmost occurrence, we are using the value of (hi-1) to be the value for the leftmost occurrence. Hence initializing hi as a.length.

Thank You!