Binary Search

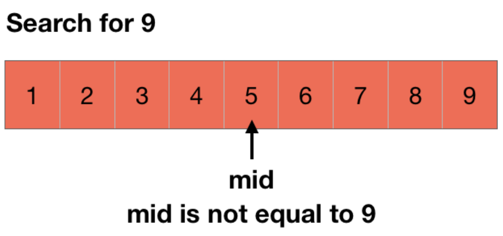
Till now, we have studied two sorting algorithms which implemented the Divide and Conquer (/course/algorithms-divide-and-conquer/) technique. Binary search is a searching algorithm which uses the Divide and Conquer technique to perform search on a sorted data.

Normally, we iterate over an array to find if an element is present in an array or not. But think about a case when the data which we are provided is a sorted one, then performing a normal search will do the work but shouldn't we extract something useful from the fact that our data is already sorted?

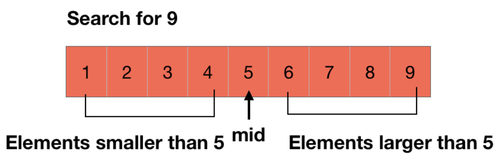
So, let's have a look at the working of the binary search and find out how to perform an optimized search on a sorted data.

Working of Binary Search

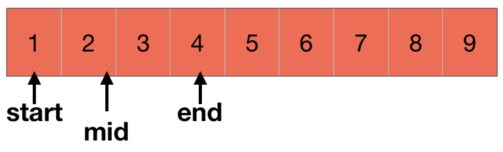
In binary search, we directly hit the middle element and then compare it with the element to be found.

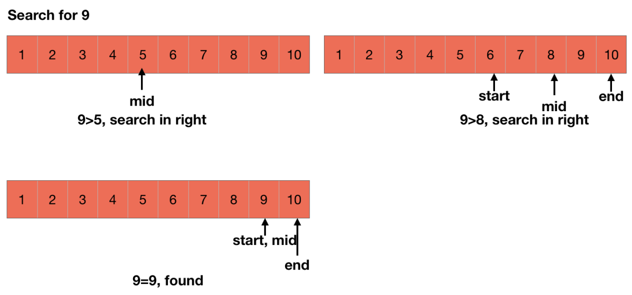


If the element to be found is equal to the middle element, then we have already found the element, otherwise, if it is smaller, then we know it is going to lie on the left side of it, else, on the right.



Let's take the case when the element to be found is smaller than the middle element, then we will repeat the same procedure on the left subarray by hitting the middle element of the left subarray.



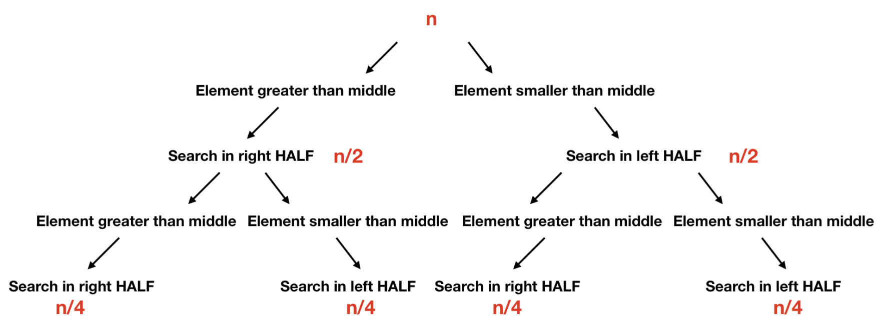


So, this is how binary search works. But before discussing the code of the binary search let's first compare binary search with the traditional search.

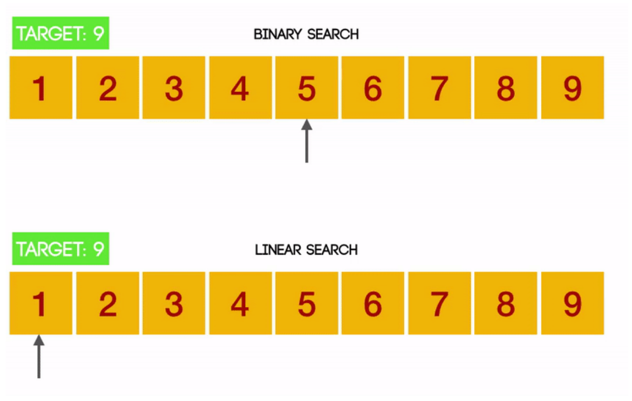
Binary Search V/S Traditional Search

In traditional search, we iterate over the array and in the worst case, it might be possible that we end up iterating over the entire array and thus, traditional search is a $O(n)$ algorithm.

Let's talk about the binary search. In binary search, without doing any analysis, we can see that the array is divided into half its initial size each time. So even in the worst case, it would end up searching only $\log_2 n$ elements.


n
Element greater than middle
Element smaller than middle
Search in right HALF
n/2
Search in left HALF
n/2
Element greater than middle
Element smaller than middle
Element greater than middle
Element smaller than middle
Search in right HALF
n/4
Search in left HALF
n/4
Search in right HALF
n/4
Search in left HALF
n/4

Thus, binary search is a $O(\lg n)$ algorithm. We are also going to mathematically see this running time later in this chapter.



Coding Binary Search

Our function is going to take an array, starting index, ending index and the element to be searched (x). So, we can declare the function for the binary search as `BINARY-SEARCH(A, start, end, x)`.

We start by hitting the middle element, so let's start by calculating the middle element first i.e., `middle = floor((start+end)/2)`.

Now, we have to compare this element with the element to be searched and if the element to be searched is the middle element, then we end up the function by returning its index i.e.,

```
if A[middle]==x
    return middle
```

If the element is smaller than the middle element, then we will repeat the binary search in the left subarray.

```
if A[middle]>x
    return BINARY-SEARCH(A, start, middle-1)
```

Similarly, we will repeat with the right subarray if the element to be found is greater than the middle element.

```
if A[middle]<x
    return BINARY-SEARCH(A, middle+1, end)
```

Thus, the entire code for the binary search can be written as:

```
BINARY-SEARCH(A, start, end, x)
  if start <= end
    middle = floor((start+end)/2)
    if A[middle]==x
      return middle

    if A[middle]>x
      return BINARY-SEARCH(A, start, middle-1, x)

    if A[middle]<x
      return BINARY-SEARCH(A, middle+1, end, x)

  return FALSE // in case, element is not in the array
```

C **Python** **Java**

```c
#include <stdio.h>

int binary_search(int a[], int start, int end, int x) {
  if(start <= end) {
    int middle = (start+end)/2;
    if(a[middle] == x) {
      return middle;
    }

    if(a[middle] > x) {
      return binary_search(a, start, middle-1, x);
    }

    if(a[middle] < x) {
      return binary_search(a, middle+1, end, x);
    }

  }
  return -1; // not found
}

int main() {
  int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int index_result = binary_search(a, 0, 9, 7);

  printf("%d\n",index_result);
  return 0;
}
```

As we are now done with the code of the binary search, let's move to its analysis.

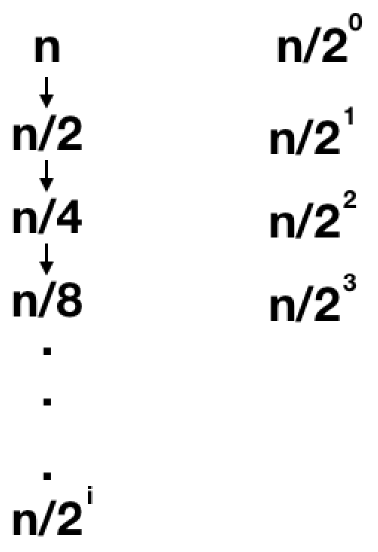
Analysis of Binary Search

In the base case, the algorithm will end up either finding the element or just failing and returning false. In both cases, the algorithm is going to take a constant time because only comparison and return statements are going to be executed.

Otherwise, comparison and calculation of the middle element will take constant time and then the problem is divided into another problem of size $\frac{n}{2}$ (either `BINARY-SEARCH(A, start, middle-1, x)` or `BINARY-SEARCH(A, middle+1, end, x)`). So, the overall running time ($T(n)$) can be written as:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

Basically, we are reducing our problem to half the size in each iteration.



In general, we can write the size of the problem as $\frac{n}{2^i}$ after doing i comparisons. Thus, for the base case, $\frac{n}{2^i} = 1 => i = \log_2 n$.

Also, the comparison will reach the base case only in the worst case and thus, binary search is a $O(\lg n)$ algorithm.

After Divide and Conquer (/course/algorithms-divide-and-conquer/), it is time to learn another algorithm which is dynamic programming. So let's continue and learn about dynamic programming.

“ You rarely win, but sometimes you do. ”

- Harper Lee

PREV (/course/algorithms-quicksort/) (/course/algorithms-dynamic-programming/) NEXT

# Further Readings

- Binary Search (/blog/article/binary-search/)