Binary Search

> Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Introduction to Binary Search

Conditions for when to apply Binary Search in a Data Structure:

To apply binary search in any data structure, the data structure must maintain the following properties:

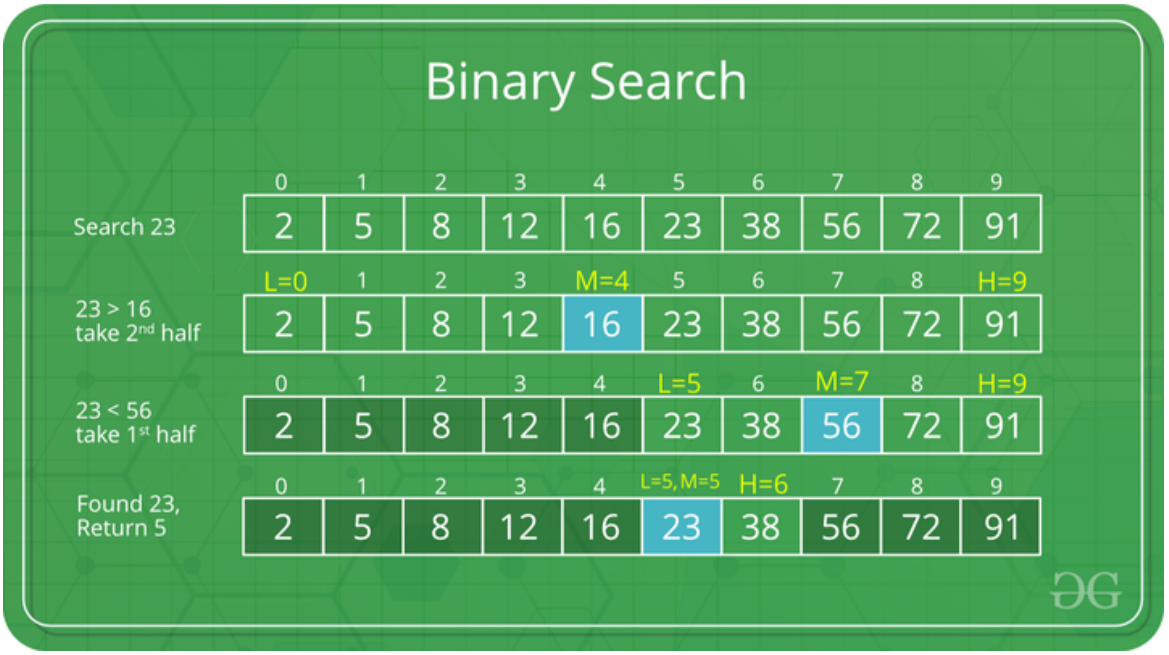
- The data structure must be sorted.
- Access to any element of the data structure takes constant time.

How does Binary Search work?

To understand the working of binary search, consider the following illustration:

> Consider an array arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, and the target = 23.

- **First Step:**
  - Initially the search space is from 0 to 9.
  - Let's denote the boundary by **L** and **H** where **L = 0** and **H = 9** initially.
  - Now mid of this search space is **M = 4**.
  - So compare **target** with **arr[M]**.
- **Second Step:**
  - As arr[4] is less than **target**, switch the search space to the right of 16, i.e., [5, 9].
  - Now **L = 5**, **H = 9** and **M** becomes **7**.
  - Compare target with **arr[M]**.
- **Third Step:**
  - arr[7] is greater than **target**.
  - Shift the search space to the left of **M**, i.e., [5, 6].
  - So, now **L = 5**, **H = 6** and **M = 6**.
  - Compare arr[M] with target.
  - Here **arr[M]** and target are the same.
- So, we have found the target.



Example of Binary Search Algorithm

When to use Binary Search?

- When searching a large dataset as it has a time complexity of O(log n), which means that it is much faster than linear search.
- When the dataset is sorted.
- When data is stored in contiguous memory.
- Data does not have a complex structure or relationships.

How to Implement Binary Search?

The basic steps to perform Binary Search are:

1. Set the low index to the first element of the array and the high index to the last element.
2. Set the middle index to the average of the low and high indices.
    - If the element at the middle index is the target element, return the middle index.
    - Otherwise, based on the value of the key to be found and the value of the middle element, decide the next search space.
        - If the target is less than the element at the middle index, set the high index to **middle index - 1**.

    - If the target is greater than the element at the middle index, set the low index to **middle index + 1**.
3. Perform step 2 repeatedly until the target element is found or the search space is exhausted.

The **Binary Search Algorithm** can be implemented in the following two ways

- Iterative Binary Search Algorithm
- Recursive Binary Search Algorithm

Given below are the pseudocodes for the approaches.

Iterative Binary Search Algorithm:

Pseudocode of Iterative Binary Search Algorithm:

> binarySearch(arr, x, low, high)
>
> repeat till low = high
>
> mid = (low + high)/2
>
> if (x = arr[mid])
>
> return mid
>
> else if (x > arr[mid])
>
> low = mid + 1
>
> else
>
> high = mid - 1

Implementation of Iterative Binary Search Algorithm:

```
// C++ program to implement iterative Binary Search
#include <bits/stdc++.h>
using namespace std;

// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
```

```
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
        r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Output

```
Element is present at index 3
```

Time Complexity: O(log N)

Auxiliary Space: O(1)

Recursive Binary Search Algorithm:

Pseudocode of Recursive Binary Search Algorithm:

> binarySearch(arr, x, low, high)
>
> if low > high
>
> return False
>
> else
>
> mid = (low + high) / 2
>
> if x = arr[mid]
>
> return mid
>
> else if x > arr[mid]
>
> return binarySearch(arr, x, mid + 1, high)
>
> else
>
> return binarySearch(arr, x, low, mid - 1)

Implementation of Recursive Binary Search Algorithm:

```
// C++ program to implement recursive Binary Search
#include <bits/stdc++.h>
using namespace std;

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
```

```
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main()
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Output

```
Element is present at index 3
```

Time Complexity: O(logN)

Auxiliary Space: O(1), If the recursive call stack is considered then the auxiliary space will be O(logN).

Binary Search Complexity Analysis:

The time and space complexities of the binary search algorithm are mentioned below.

Time Complexity of Binary Search Algorithm:

- **Best Case:** O(1)

> Best case is when the element is at the middle index of the array. It takes only one comparison to find the target element. So the best case complexity is O(1).

- **Average Case:** O(log N)

> Consider array arr[] of length N and element X to be found. There can be two cases:
>
> - Case1: Element is present in the array
> - Case2: Element is not present in the array.
>
> There are **N** Case1 and **1** Case2. So total number of cases = **N+1**. Now notice the following:
>
> - An element at index N/2 can be found in **1** comparison
> - Elements at index N/4 and 3N/4 can be found in **2** comparisons.
> - Elements at indices N/8, 3N/8, 5N/8 and 7N/8 can be found in **3** comparisons and so on.
>
> Based on this we can conclude that elements that require:
>
> - 1 comparison = 1
> - 2 comparisons = 2
> - 3 comparisons = 4

> - **x** comparisons = **2x-1** where **x** belongs to the range **[1, logN]** because maximum comparisons = maximum time N can be halved = maximum comparisons to reach 1st element = logN.
>
> So, total comparisons
>
> = 1*(elements requiring 1 comparisons) + 2*(elements requiring 2 comparisons) + . . . + logN*(elements requiring logN comparisons)
>
> = 1*1 + 2*2 + 3*4 + . . . + logN * (2logN-1)
>
> = 2logN * (logN - 1) + 1
>
> = **N * (logN - 1) + 1**
>
> Total number of cases = **N+1**.
>
> Therefore, the average complexity = (**N*(logN - 1) + 1)/N+1** = **N*logN / (N+1) + 1/(N+1)**. Here the dominant term is N*logN/(N+1) which is approximately **logN**. So the average case complexity is **O(logN)**

- **Worst Case:** O(log N)

> The worst case will be when the element is present in the first position. As seen in the average case, the comparison required to reach the first element is logN. So the time complexity for the worst case is O(logN).

Auxiliary Space Complexity of Binary Search Algorithm

- **O(1)** as no extra space is used

Correct way to calculate "mid" in Binary Search:

Here to calculate "**mid**" we are doing the following:

```
int mid = low + (high - low)/2;
```

Maybe, you wonder why we are calculating the ***middle index*** this way, we can simply add the **lower** and **higher** index and divide it by 2 like

> int mid = (low + high)/2;

But if we calculate the **middle index** like this, it fails for larger values of **int** variables. Specifically, it fails if the sum of **low** and **high** is greater than the maximum positive **int** value(231 - 1). The sum overflows to a negative value and the value stays negative when divided by 2.

So it's better to calculate "**mid**" in the way it is done in the above code.

Advantages of Binary Search:

- Binary search is faster than linear search, especially for large arrays. As the size of the array increases, the time it takes to perform a linear search increases linearly, while the time it takes to perform a binary search increases logarithmically.
- Binary search is more efficient than other searching algorithms that have a similar time complexity, such as interpolation search or exponential search.
- Binary search is relatively simple to implement and easy to understand, making it a good choice for many applications.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.
- Binary search can be used as a building block for more complex algorithms, such as those used in computer graphics and machine learning.

Drawbacks of Binary Search:

- We require the array to be sorted. If the array is not sorted, we must first sort it before performing the search. This adds an additional O(N * logN) time complexity for the sorting step, which makes it irrelevant to use binary search.
- Binary search requires that the data structure being searched be stored in contiguous memory locations. This can be a problem if the data structure is too large to fit in memory, or if it is stored on external memory such as a hard drive or in the cloud.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered. This can be a problem if the elements of the array are not naturally ordered, or if the ordering is not well-defined.
- Binary search can be less efficient than other algorithms, such as hash tables, for searching very large datasets that do not fit in memory.

Applications of Binary Search:

- Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- Commonly used in Competitive Programming.
- Can be used for searching in computer graphics. Binary search can be used as a building block for more complex algorithms used in computer graphics, such as algorithms for ray tracing or texture mapping.
- Can be used for searching a database. Binary search can be used to efficiently search a database of records, such as a customer database or a product catalog.

Conclusion:

To wrap it up, it can be said that Binary search is an efficient algorithm for finding an element within a sorted array. The time complexity of the binary search is O(log n). Though, one of the main drawbacks of binary search is that the array must be sorted. this is a useful algorithm for building more complex algorithms in computer graphics and machine learning or several other cases.