Binary Search

Binary search is a highly efficient algorithm used to find an element's position in a sorted array. Unlike linear search, which checks each element one by one, binary search divides the search space in half with each step, making it faster. Here's a detailed explanation:

1. Prerequisites

- **Sorted Array**: Binary search can only be applied to a sorted array. If the array is not sorted, the results will be unpredictable.
- **Access to Middle Element**: The algorithm requires the ability to access the middle element directly, so random access is essential (like in arrays or lists).

2. How It Works

Binary search follows a divide-and-conquer approach. Here are the steps:

1. **Initialize Pointers**: Start with two pointers: `low` (pointing to the first element of the array) and `high` (pointing to the last element of the array).
2. **Find the Midpoint**: Calculate the middle index:
   mid=low+2high−low

   mid=low+high−low2\text{mid} = \text{low} + \frac{\text{high} - \text{low}}{2}
3. **Compare the Middle Element**:
   - If the middle element equals the target, you've found the element, and the search ends.
   - If the middle element is less than the target, move the `low` pointer to `mid + 1` (discarding the left half).
   - If the middle element is greater than the target, move the `high` pointer to `mid - 1` (discarding the right half).
4. **Repeat Until Found or Exceeded**: Continue this process until the target is found or the `low` pointer exceeds the `high` pointer (meaning the target is not present in the array).

3. Example

Suppose you want to find the number `7` in the following sorted array:

```
csharp
Copy code
[1, 3, 5, 7, 9, 11, 13, 15]
```

1. **Initial Setup**:
   - `low = 0` (index of `1`)
   - `high = 7` (index of `15`)
2. **Step 1**:
   - Calculate `mid`: `mid = 0 + (7 - 0) / 2 = 3`
   - Array element at index `3` is `7`, which matches the target.
   - **Found!**

The algorithm successfully finds `7` in just one step, demonstrating how efficient binary search is.

4. Complexity Analysis

- **Time Complexity**:
  - **Best Case**: O(1) — When the middle element is the target.

    O(1)O(1)
  - **Average and Worst Case**: O(logn) — Each iteration reduces the search space by half.

    O(logn)O(\log n)
- **Space Complexity**:
  - **Iterative Approach**: O(1) — Uses constant extra space.

    O(1)O(1)
  - **Recursive Approach**: O(logn) — Due to the recursion stack.

O(logn)O(\log n)

5. Binary Search Implementation

Here’s an example implementation in Java:

Iterative Approach

```
public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            // Check if target is present at mid
            if (arr[mid] == target) {
                return mid;
            }

            // If target is greater, ignore the left half
            if (arr[mid] < target) {
                low = mid + 1;
            }

            // If target is smaller, ignore the right half
            else {
                high = mid - 1;
            }
        }

        // Target not found
        return -1;
    }
```

```
}
```

Recursive Approach

```
public class BinarySearchRecursive {
    public static int binarySearch(int[] arr, int low, int hi
gh, int target) {
        if (low <= high) {
            int mid = low + (high - low) / 2;

            // Check if the target is present at mid
            if (arr[mid] == target) {
                return mid;
            }

            // If target is smaller, search in the left half
            if (arr[mid] > target) {
                return binarySearch(arr, low, mid - 1, targe
t);
            }

            // Otherwise, search in the right half
            return binarySearch(arr, mid + 1, high, target);
        }

        // Target not found
        return -1;
    }
}
```

6. Use Cases of Binary Search

- **Finding an Element in a Sorted Array**
- **Searching in a Dictionary or Phonebook**
- **Efficiently Finding Lower or Upper Bound**
- **Applications in Game Development (e.g., hit detection)**
- **Algorithm Optimization (e.g., searching for the best value that satisfies a condition)**

7. Pros and Cons

Pros:

- **Efficient**: Much faster than linear search, especially for large data sets.
- **Simple to Implement**: Easy to code both iteratively and recursively.

Cons:

- **Requires Sorted Data**: Can't be used directly on unsorted arrays.
- **Random Access Needed**: May not be suitable for data structures that don't allow direct access (e.g., linked lists).

Binary search is a fundamental algorithm, crucial for understanding more complex searching and optimization techniques.