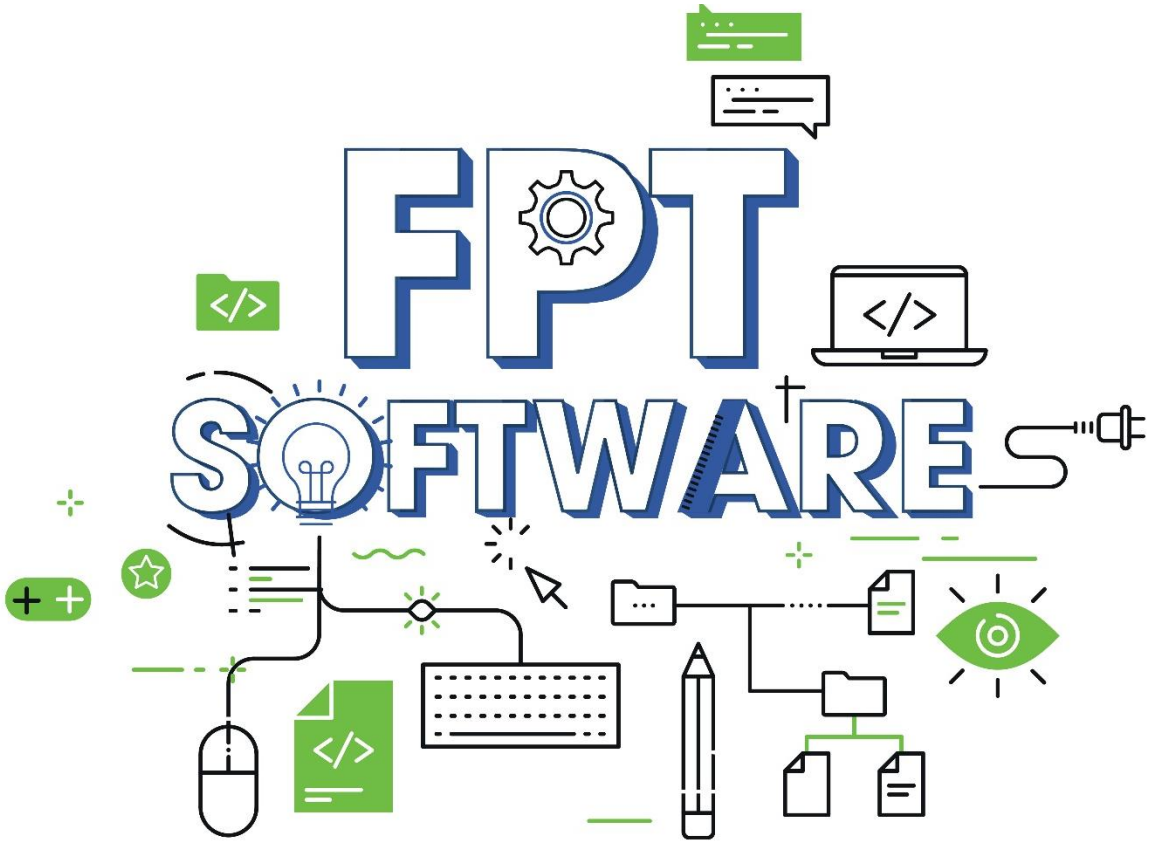
Basic Searching Algorithms

Fsoft Academy



Lesson Objectives

- **Understanding** different search algorithms: linear search, binary search
- **Analyzing** the performance of search algorithms
- **Implementing** basic search algorithms
- **Applying** search algorithms to real-world problems

Agenda

1. Searching Algorithms
2. Linear Search
3. Binary Search
4. Q&A

Section 1

SEARCHING ALGORITHMS

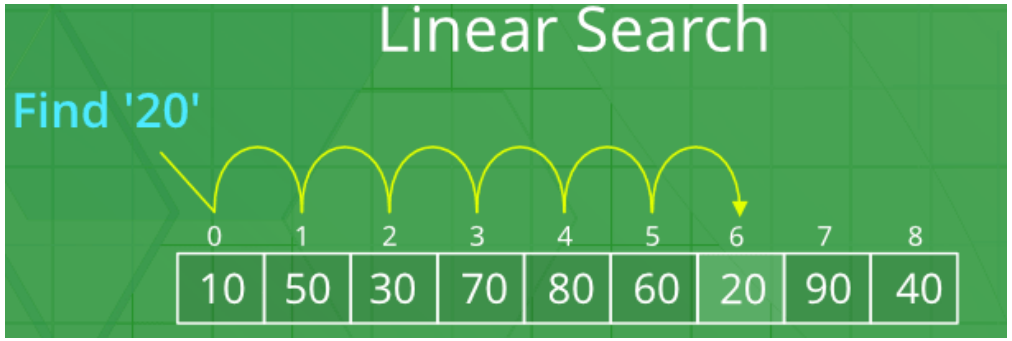
Introduction

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

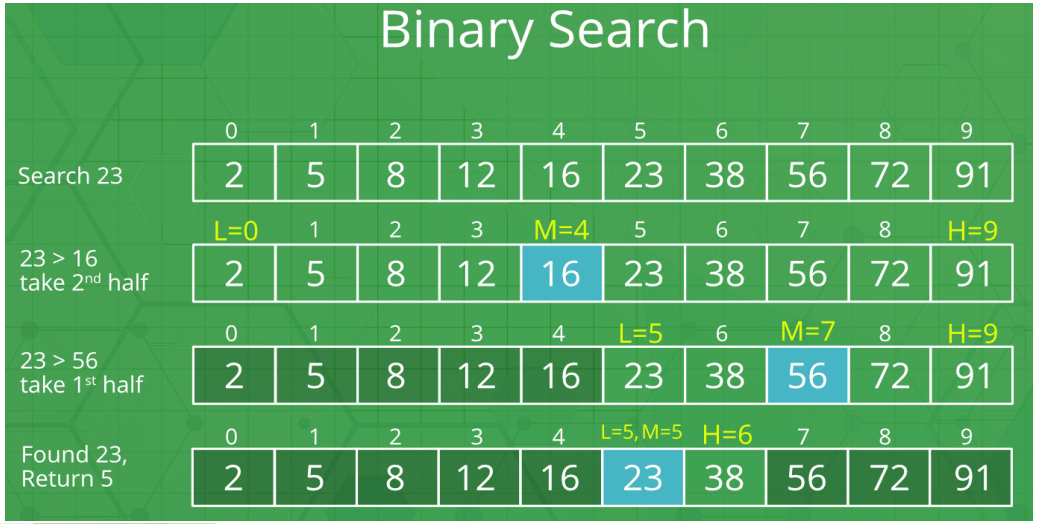
- Based on the type of search operation, these algorithms are generally classified into two categories:
  - ✓ **Sequential Search or Linear Search**: In this, the list or array is *traversed sequentially and every element is checked*.
  - ✓ **Interval Search or Binary Search**: These algorithms are specifically designed for *searching in sorted data-structures*. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half.

Linear Search and Binary Search

- **Linear Search** to find the element “20” in a given list of numbers



- **Binary Search** to find the element “23” in a given list of numbers



Section 2

LINEAR SEARCH

Overview

- **Linear search is a very simple search algorithm.**
  - ✓ *Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.*
- **Algorithm**

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Pseudocode

```
procedure linear_search (list, value)

    for each item in the list
        if match item == value
            return the item's location
        end if
    end for

end procedure
```

Linear Search Program in Java

```
static final int MAX = 20;

// array of items on which linear search will be conducted.
static int[] intArray = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};
```

Linear Search Program in Java

```java
// this method makes a linear search.
static int find(int data) {

   int comparisons = 0;
   int index = -1;

   // navigate through all items
   for(int i = 0; i < MAX; i++) {
      // count the comparisons made
      comparisons++;

      // if data found, break the loop
      if(data == intArray[i]) {
         index = i;
         break;
      }
   }
   System.out.printf("Total comparisons made: %d", comparisons);
   return index;
} // end find()
```

```java
static void display() {
   System.out.print("[");

   // navigate through all items
   for(int i = 0; i < MAX; i++) {
      System.out.printf("%d ",intArray[i]);
   }

   System.out.println("]");
} // end display()
```

Linear Search Program in Java

```
public static void main(String[] args) {
    System.out.print("Input Array: ");
    display();

    // find location of 1
    int location = find(55);

    // if element was found
    if (location != -1) {
        System.out.printf("%nElement found at location: %d", (location + 1));
    } else {
       System.out.printf("%nElement not found.");
    }
} // end main()
```

- **Output**

```
Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66 ]

Total comparisons made: 19
Element found at location: 19
```

Complexity of Linear Search

- As linear search scans each element one by one *until the element is not found*.
- If the number of elements increases, *the number of elements to be scanned is also increased*.
- We can say that the ***time taken to search the elements is proportional to the number of elements***.
  - ✓ **Time Complexity:** O(n), where n is the size of the input array. The worst-case scenario is when the target element is not present in the array, and the function has to go through the entire array to figure that out.
  - ✓ **Auxiliary Space:** O(1), the function uses only a constant amount of extra space to store variables. The amount of extra space used does not depend on the size of the input array.

Section 3

BINARY SEARCH

Overview

Binary search algorithm works on the *principle of divide and conquer*. For this algorithm to work properly, the data collection should be in the **sorted form**.

- **Binary search** is a fast search algorithm with run-time complexity of **O(log n).**
- **Binary search** looks for a particular item by comparing the middle most item of the collection.
  - ✓ If a match occurs, then the index of item is returned.
  - ✓ If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
  - ✓ Otherwise, the item is searched for in the sub-array to the right of the middle item.
  - ✓ This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Work?

- For a binary search to work, it is mandatory for the target **array to be sorted**.
- Let us assume that we need to **search the location of value 31** using binary search.

- First, we shall determine half of the array by using this formula −

$$mid = low + (high - low) / 2$$

- Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.

How Binary Search Work?

- Now we **compare the value stored at location 4**, with the value being searched, i.e. 31.
- We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

- We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

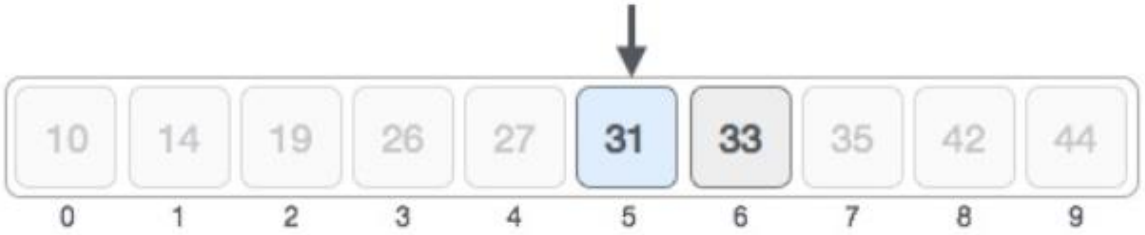
How Binary Search Work?

- **Our new mid is 7 now.** We compare the value stored at location 7 with our target value 31.

- The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

How Binary Search Work?

- Hence, we calculate the mid again. This time it is 5.



- We compare the value stored at location 5 with our target value. We find that it is a match.

- We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

- The pseudocode of binary search algorithms should look like this

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound +
        ( upperBound - lowerBound ) / 2
```

```
if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at
                     location midPoint
   end while

end procedure
```

Implementation in Java

```
static final int MAX = 20;

// array of items on which linear search will be conducted.
static int[] intArray = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};
```

Implementation in Java

```
static int find(int data) {
   int lowerBound = 0;
   int upperBound = MAX - 1;
   int midPoint = -1;
   int comparisons = 0;
   int index = -1;

   while (lowerBound <= upperBound) {
      System.out.printf("Comparison %d\n", (comparisons + 1));
      System.out.printf("lowerBound : %d, intArray[%d] = %d\n", lowerBound, lowerBound, intArray[lowerBound]);
      System.out.printf("upperBound : %d, intArray[%d] = %d\n", upperBound, upperBound, intArray[upperBound]);
      comparisons++;
      // compute the mid point
      // midPoint = (lowerBound + upperBound) / 2;
      midPoint = lowerBound + (upperBound - lowerBound) / 2;

      // data found
      if (intArray[midPoint] == data) {
         index = midPoint;
         break;
      }
```

Implementation in Java

```
      else {
         // if data is larger
         if (intArray[midPoint] < data) {
            // data is in upper half
            lowerBound = midPoint + 1;
         }
         // data is smaller
         else {
            // data is in lower half
            upperBound = midPoint - 1;
         }
      }
}

System.out.printf("Total comparisons made: %d",
                                 comparisons);
return index;

} // end find()
```

```
static void display() {
   System.out.print("[");

   // navigate through all items
   for (int i = 0; i < MAX; i++) {
       System.out.printf("%d ", intArray[i]);
   }

   System.out.println("]");
} // end display()
```

Implementation in Java

```
public static void main(String[] args) {

   System.out.print("Input Array: ");
   display();

   // find location of 1
   int location = find(55);

   // if element was found
   if (location != -1) {
      System.out.printf("%nElement found at location: %d", (location + 1));
   } else {
      System.out.printf("%nElement not found.");
   }
}
```

Implementation in Java

- **Output:**

```
Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66 ]

Comparison 1
lowerBound : 0, intArray[0] = 1
upperBound : 19, intArray[19] = 66
Comparison 2
lowerBound : 10, intArray[10] = 15
upperBound : 19, intArray[19] = 66
Comparison 3
lowerBound : 15, intArray[15] = 34
upperBound : 19, intArray[19] = 66
Comparison 4
lowerBound : 18, intArray[18] = 55
upperBound : 19, intArray[19] = 66
Total comparisons made: 4
Element found at location: 19
```

Lesson Summary

- Searching Algorithms
- Linear Search
- Binary Search

THANK YOU!