



جامعة المستقبل  
AL MUSTAQBAL UNIVERSITY



قسم الأمن الإلكتروني

**DEPARTMENT OF CYBER SECURITY**

**SUBJECT:**

**SEARCHING AND SORTING ALGORITHMS**

**CLASS:**

**SECOND**

**LECTURER: M.SC.MUNTATHER AL-MUSSAWEE**

**LECTURE: (1)**

**INTRODUCTION**

There are basically **two aspects** of computer programming. ***One is data organization*** also commonly called as **data structures**. Till now we have seen about data structures and the techniques and algorithms used to access them. The ***other part*** of computer programming involves **choosing the appropriate algorithm** to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This lecture introduces this important aspect of problem solving.

## INTRODUCTION TO SEARCHING AND SORTING ALGORITHMS

تعتبر خوارزمية البحث ركنا أساسيا من أركان **علم الخوارزميات** وتتخذ هذه الخوارزمية عدة أشكال، من أبسطها البحث عن عدد في مصفوفة مُحددة الحجم ويزداد الأمر تعقيدا عند الانتقال إلى البحث عن كلمة داخل نص، تماما كما ترى في محررات النصوص العادية والتي تحتوي على خاصية **Find & Replace** حيث أن أغلب المحررات الحالية تستخدم خوارزميه بحث تسمى

**Boyer-Moore Searching** التي تُعد تقريبا من أسرع الخوارزميات في مجال البحث.

هناك أيضا بحث من نوع آخر، فماذا لو كانت لدينا مجموعة حروف

ونريد إيجاد جميع الكلمات التي تبدأ بهذه الحروف؟؟ عادة ما يُسمى هذا النوع من الخوارزميات بـ

**prefix searching** ولهذا النوع تطبيقات كثيرة خصوصا في محركات البحث والقواميس والمتصفحات التي تستخدم هذه الخوارزمية عند كتابتك لموقع يبدأ بحرف كنت قد زرته سابقا.

لا تقتصر خوارزمية البحث على ما ذكرناه آنفا، فهناك نوع آخر من الخوارزميات يُستخدم لإيجاد نص قريب

من النص الذي كنت تبحث عنه، حيث تقوم الخوارزمية بالبحث عن 4 أو 5 كلمات قريبة من الكلمات الخاطئة، تماما كما يفعل **Google** عند الترجمة أو **Office Word** عند كتابة نص يحتوي على أخطاء إملائية وتعتمد أغلب هذه الخوارزميات على الوزن الصوتي للحرف ومن أشهرها **Soundex Searching**

ليس هذا فقط، **مضادات الفيروسات** تستخدم خوارزميات بحث سريعة للبحث عن وجود توقيع مطابق لأحد بيانات الملف المراد فحصه، وبما أن قواعد بيانات التوقيعات تكون ضخمة للغاية فالبحث عن كل توقيع



سيكون بطيئاً جداً وهناك الأفضل، حيث توجد خوارزميات بإمكانها البحث عن عدة توافيق في نفس اللحظة وتستخدم بنية شجرية للقيام بهذا الأمر، هناك أيضاً خوارزميات أخرى تعتمد على مفاهيم مختلفة مثل **Hash Table** وعدة أمور أخرى، ويُسمى هذا النوع

من الخوارزميات بـ **Multiple pattern searching**

وهو من أصعب الخوارزميات، وهناك العديد من مضادات الفيروسات التي تستخدم مثل هذه الخوارزميات مثل **ClamAV**.

**Searching** is an operation or a technique that helps find the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching techniques that are being followed in data structure are listed below:

1. Linear or sequential search
2. Binary search

**Sorting** allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words are arranged in alphabetical order and telephone directory in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:



## 1. Internal sorting

## 2. External sorting

If all the elements to be sorted are present in the main memory, then such sorting is called internal sorting on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called external sorting. Here we study only internal sorting techniques.

### 1. LINEAR SEARCH:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list, then the search is successful otherwise unsuccessful.

Suppose there are “n” elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need  $[(n+1)/2]$  comparisons to search an element. If search is not successful, you would need “n” comparisons.

The time complexity of linear search is  $O(n)$ .



## Algorithm

```
LinearSearch ( 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
```

Here is a simple C++ program that demonstrates sequential search (also known as linear search). This search algorithm checks each element in a list (or array) one by one to find the target value.

### Sequential Search Example ▾

```
#include <iostream>
using namespace std;

// Function for sequential search
int sequentialSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return index if target is found
        }
    }
    return -1; // Return -1 if target is not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50}; // Example array
    int target;

    cout << "Enter the value to search: ";
    cin >> target;

    int size = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array
    int result = sequentialSearch(arr, size, target);

    if (result != -1) {
        cout << "Element found at index: " << result << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```



### Example 1:

Let us illustrate linear search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for  $x = 7$ , Search successful, data found at 3rd position.
2. Searching for  $x = 82$ , Search successful, data found at 7th position.
3. Searching for  $x = 42$ , Search un-successful, data not found.

**Example 2:** Let us take an example of an array  $A[7]=\{5,2,1,6,3,7,8\}$ . Array A has 7 items. Let us assume we are looking for 7 in the array. Targeted item=7.

Here, we have

$A[7]=\{5,2,1,6,3,7,8\}$

Index	0	1	2	3	4	5	6
Element	5	2	1	6	3	7	8

$X=7$

At first, When  $i=0$  ( $A[0]=5$ ;  $X=7$ ) not matched

$i++$  now,  $i=1$  ( $A[1]=2$ ;  $X=7$ ) not matched

$i++$  now,  $i=2$  ( $A[2]=1$ ;  $X=7$ ) not matched

...

....

$i++$  when,  $i=5$  ( $A[5]=7$ ;  $X=7$ ) **Match Found**

Hence, Element  $X=7$  found at index 5.



## ❖ A RECURSIVE PROGRAM FOR LINEAR SEARCH

Here is a recursive C++ program to perform linear (sequential) search:

### Recursive Linear Search ▾

```
#include <iostream>
using namespace std;

// Recursive function for linear search
int recursiveLinearSearch(int arr[], int size, int target, int index = 0) {
    // Base case: if index exceeds the size of the array, target is not found
    if (index >= size) {
        return -1;
    }
    // Check if the current element matches the target
    if (arr[index] == target) {
        return index;
    }
    // Recursive call to check the next element
    return recursiveLinearSearch(arr, size, target, index + 1);
}

int main() {
    int arr[] = {10, 20, 30, 40, 50}; // Example array
    int target;

    cout << "Enter the value to search: ";
    cin >> target;

    int size = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array
    int result = recursiveLinearSearch(arr, size, target);

    if (result != -1) {
        cout << "Element found at index: " << result << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```





### ❖ EXPLANATION:

1. **Base Case:**
  - If the index reaches or exceeds the size of the array, it means the target is not present, and `-1` is returned.
2. **Recursive Case:**
  - If the current element (`arr[index]`) matches the target, return the index.
  - Otherwise, recursively call the function with `index + 1` to check the next element.
3. **Main Function:**
  - The user inputs a target value to search in the array. The recursive function is called with the initial index set to `0`.

### ❖ EXAMPLE EXECUTION:

If the array is `{10, 20, 30, 40, 50}` and the user searches for `30`:

- The function will check elements at indices `0`, `1`, and `2`, and then return `2` as the result.

## 2. BINARY SEARCH

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
  2. If we get a match, we return the index of the middle element.
  3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
  4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element (as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
  5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.
-





Binary Search is useful when there are large number of elements in an array and they are sorted. So a necessary condition for Binary search to work is that the list/array should be sorted.

### ❖ FEATURES OF BINARY SEARCH

1. It is great to search through large sorted arrays.
2. It has a time complexity of  **$O(\log n)$**  which is a very good time complexity. It has a simple implementation.

Binary search is a fast search algorithm with run-time complexity of  $O(\log n)$ . This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form.

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 sub array reduces to zero.

### ❖ HOW BINARY SEARCH WORKS?

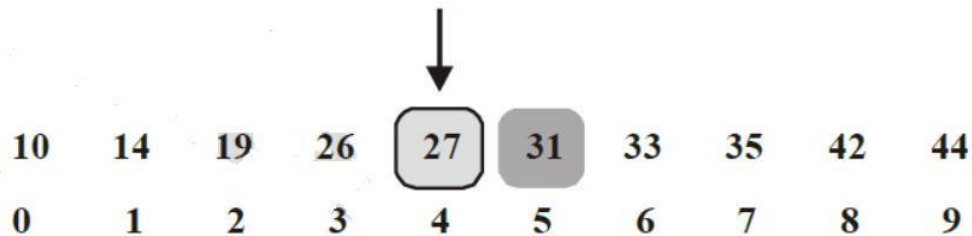
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value **31** using binary search.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

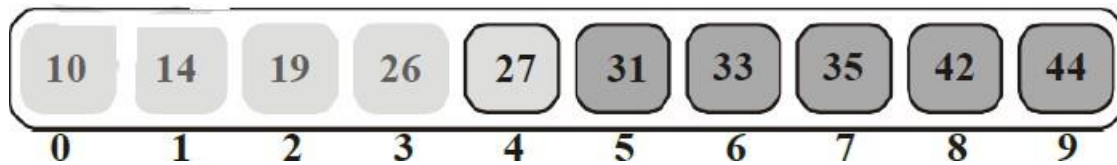
First, we shall determine half of the array by using this formula –

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

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



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.

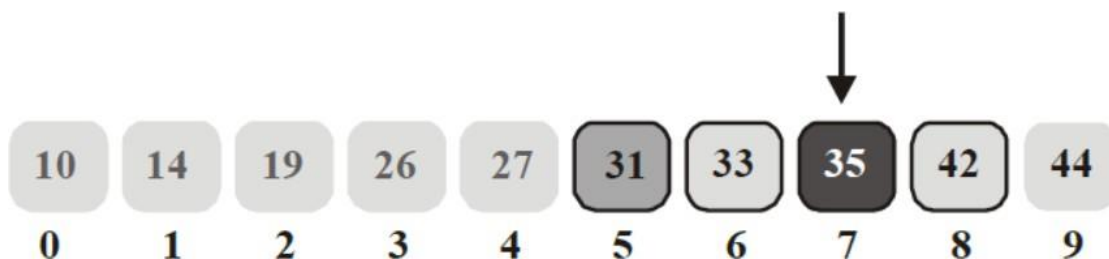


We change our low to mid + 1 and find the new mid value again.

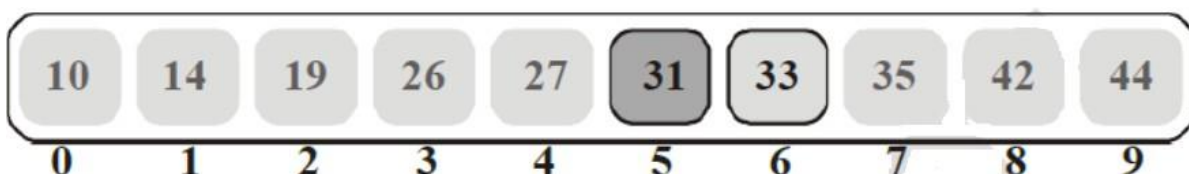
**low = mid + 1**

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

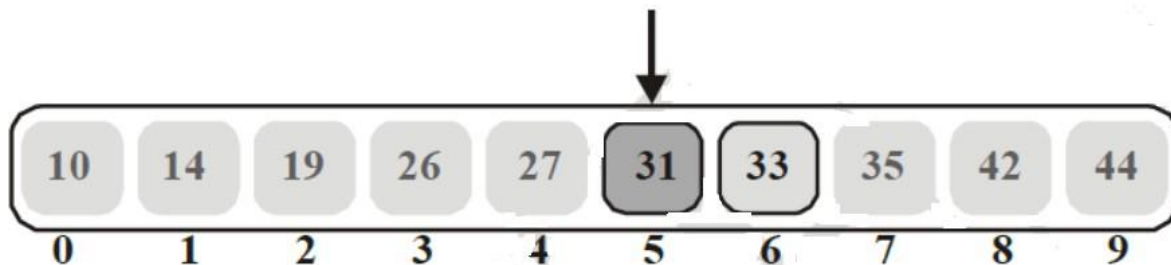
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



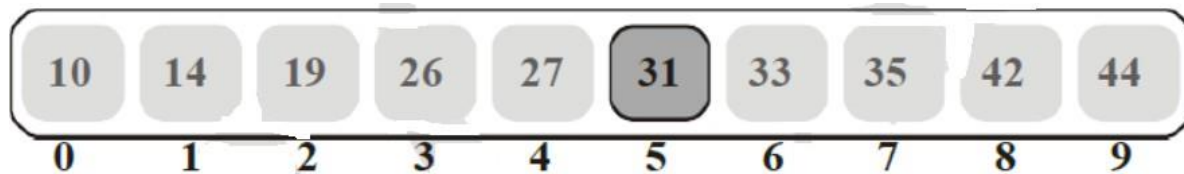
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



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.

### ❖ EXAMPLE 1:

Let us illustrate binary search on the following 12 elements:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Elements	4	7	8	9	16	20	24	38	39	45	54	77

**If we are searching for  $x = 4$ : (This needs 3 comparisons)**

low = 1, high = 12, mid =  $13/2 = 6$ , check 20

low = 1, high = 5, mid =  $6/2 = 3$ , check 8

low = 1, high = 2, mid =  $3/2 = 1$ , check 4, found



**If we are searching for  $x = 7$ : (This needs 4 comparisons)**

low = 1, high = 12, mid =  $13/2 = 6$ , check 20

low = 1, high = 5, mid =  $6/2 = 3$ , check 8

low = 1, high = 2, mid =  $3/2 = 1$ , check 4

low = 2, high = 2, mid =  $4/2 = 2$ , check 7, found

**If we are searching for  $x = 8$ : (This needs 2 comparisons)**

low = 1, high = 12, mid =  $13/2 = 6$ , check 20

low = 1, high = 5, mid =  $6/2 = 3$ , check 8, found

**If we are searching for  $x = 9$ : (This needs 3 comparisons)**

low = 1, high = 12, mid =  $13/2 = 6$ , check 20

low = 1, high = 5, mid =  $6/2 = 3$ , check 8

low = 4, high = 5, mid =  $9/2 = 4$ , check 9, found

**If we are searching for  $x = 16$ : (This needs 4 comparisons)**

low = 1, high = 12, mid =  $13/2 = 6$ , check 20

low = 1, high = 5, mid =  $6/2 = 3$ , check 8

low = 4, high = 5, mid =  $9/2 = 4$ , check 9

low = 5, high = 5, mid =  $10/2 = 5$ , check 16, found

**If we are searching for  $x = 20$ : (This needs 1 comparison)**

low = 1, high = 12, mid =  $13/2 = 6$ , check 20, found

**❖ AN ITERATIVE BINARY SEARCH PROGRAM**

Here is a C++ program for binary search. Binary search is a more efficient search algorithm compared to linear search, but it requires the array to be sorted.



## Binary Search ▾

```
#include <iostream>
using namespace std;

// Function for binary search
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2; // Calculate mid to prevent overflow

        // Check if target is at mid
        if (arr[mid] == target) {
            return mid;
        }
        // If target is smaller than mid, search in the left subarray
        else if (arr[mid] > target) {
            right = mid - 1;
        }
        // If target is larger than mid, search in the right subarray
        else {
            left = mid + 1;
        }
    }
    return -1; // Return -1 if target is not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50}; // Sorted array
    int target;

    cout << "Enter the value to search: ";
    cin >> target;

    int size = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array
    int result = binarySearch(arr, size, target);

    if (result != -1) {
        cout << "Element found at index: " << result << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```



## ❖ RECURSIVE BINARY SEARCH:

### Recursive Binary Search ▾

```
#include <iostream>
using namespace std;

// Recursive function for binary search
int recursiveBinarySearch(int arr[], int left, int right, int target) {
    // Base case: If left index exceeds right, target is not found
    if (left > right) {
        return -1;
    }

    int mid = left + (right - left) / 2; // Calculate mid to prevent overflow

    // Check if target is at mid
    if (arr[mid] == target) {
        return mid;
    }
    // If target is smaller than mid, search in the left subarray
    else if (arr[mid] > target) {
        return recursiveBinarySearch(arr, left, mid - 1, target);
    }
    // If target is larger than mid, search in the right subarray
    else {
        return recursiveBinarySearch(arr, mid + 1, right, target);
    }
}

int main() {
    int arr[] = {10, 20, 30, 40, 50}; // Sorted array
    int target;

    cout << "Enter the value to search: ";
    cin >> target;

    int size = sizeof(arr) / sizeof(arr[0]); // Calculate the size of the array
    int result = recursiveBinarySearch(arr, 0, size - 1, target);

    if (result != -1) {
        cout << "Element found at index: " << result << endl;
    } else {
        cout << "Element not found in the array." << endl;
    }

    return 0;
}
```



## ❖ EXPLANATION:

### 1. Binary Search (Iterative and Recursive):

- Works by dividing the search space into halves.
- Continually compares the middle element to the target.
- Eliminates half the search space in each step.
- Stops when the target is found or the search space becomes empty.

### 2. Main Differences:

- **Iterative:** Uses a loop to perform the search.
- **Recursive:** Calls itself with updated bounds (`left` and `right`).

### 3. Input Requirement:

- The array must be sorted for binary search to work.