جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

قسم الامن السيبراني

DEPARTMENT OF CYBER SECURITY

SUBJECT:

SEARCHING AND SORTING ALGORITHMS

CLASS:

SECOND

LECTURER:

ASST. PROF. DR. ALI KADHUM AL-QURABY

LECTURE: (1)

INTRODUCTION

❖ 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 Ï(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 –

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.

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

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.

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

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

low = mid + 1

mid = low + (high - low) / 2

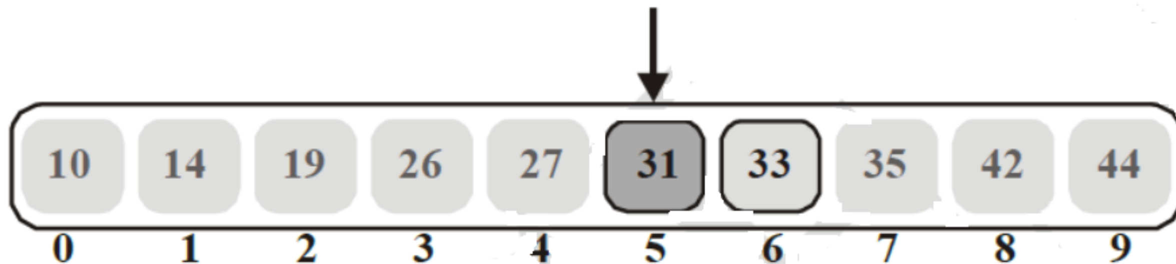
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

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

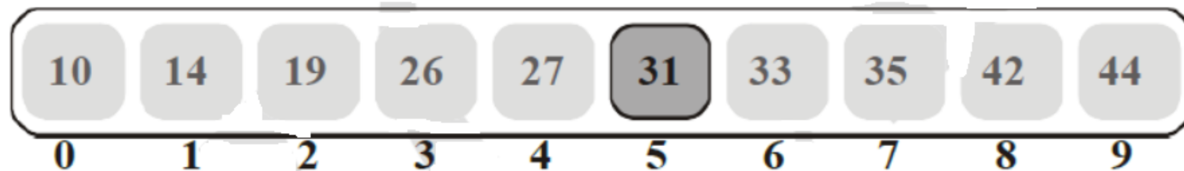
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

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

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

```cpp
#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.