University American College Skopje

SINCE 2005

DATA STRUCTURES AND ALGORITHMS

- LECTURE 4-

(2ND YEAR OF STUDY)

Skopje, 2023/24

Contents

4.1. Introduction

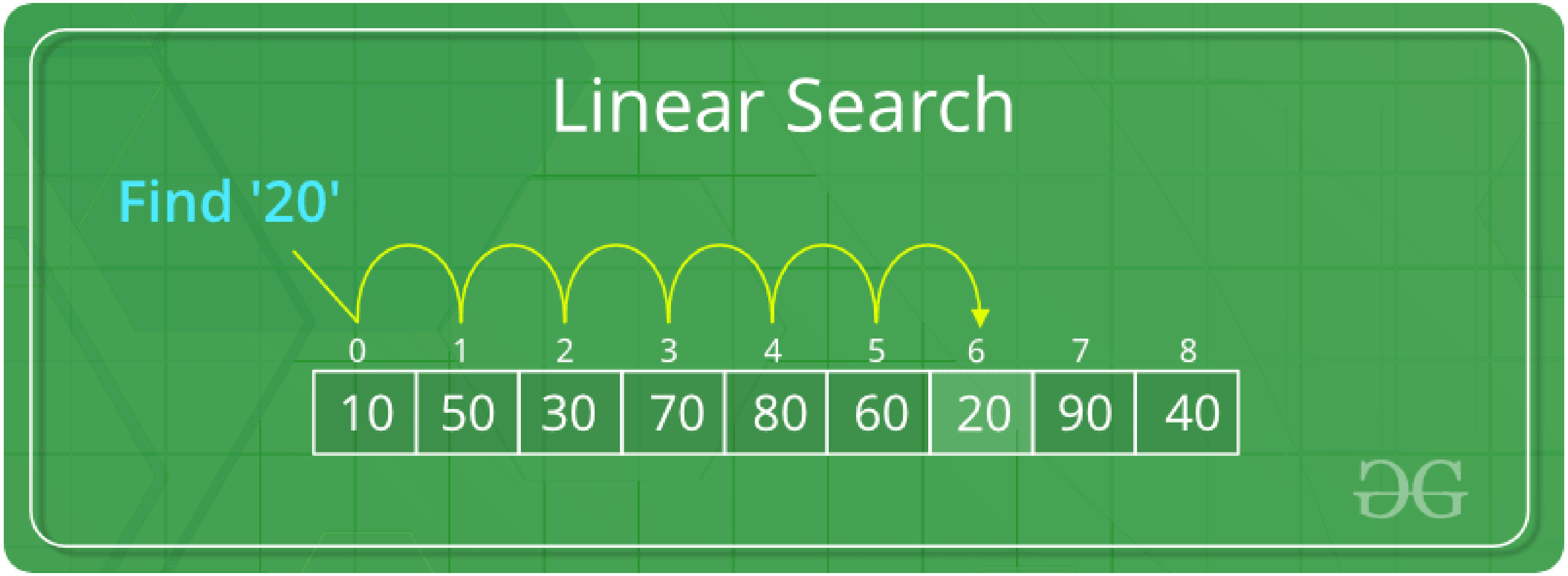
- **Searching Algorithms** are designed to check for an element, or retrieve an element from any data structure where it is stored.
- These algorithms are crucial for tasks, like: searching for a particular record in a database, finding an element in a sorted list, or locating a file on a computer.
- Although *search engines* use search algorithms, they belong to the study of *information retrieval*, not algorithms.

4.1. Introduction

- Based on the type of search operation, these algorithms are generally classified into 2 categories:
  - **Sequential Search:** In this, the list or array is traversed sequentially and every element is checked. For example: ***Linear Search.***
  - **Interval 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. For Example: ***Binary Search.***

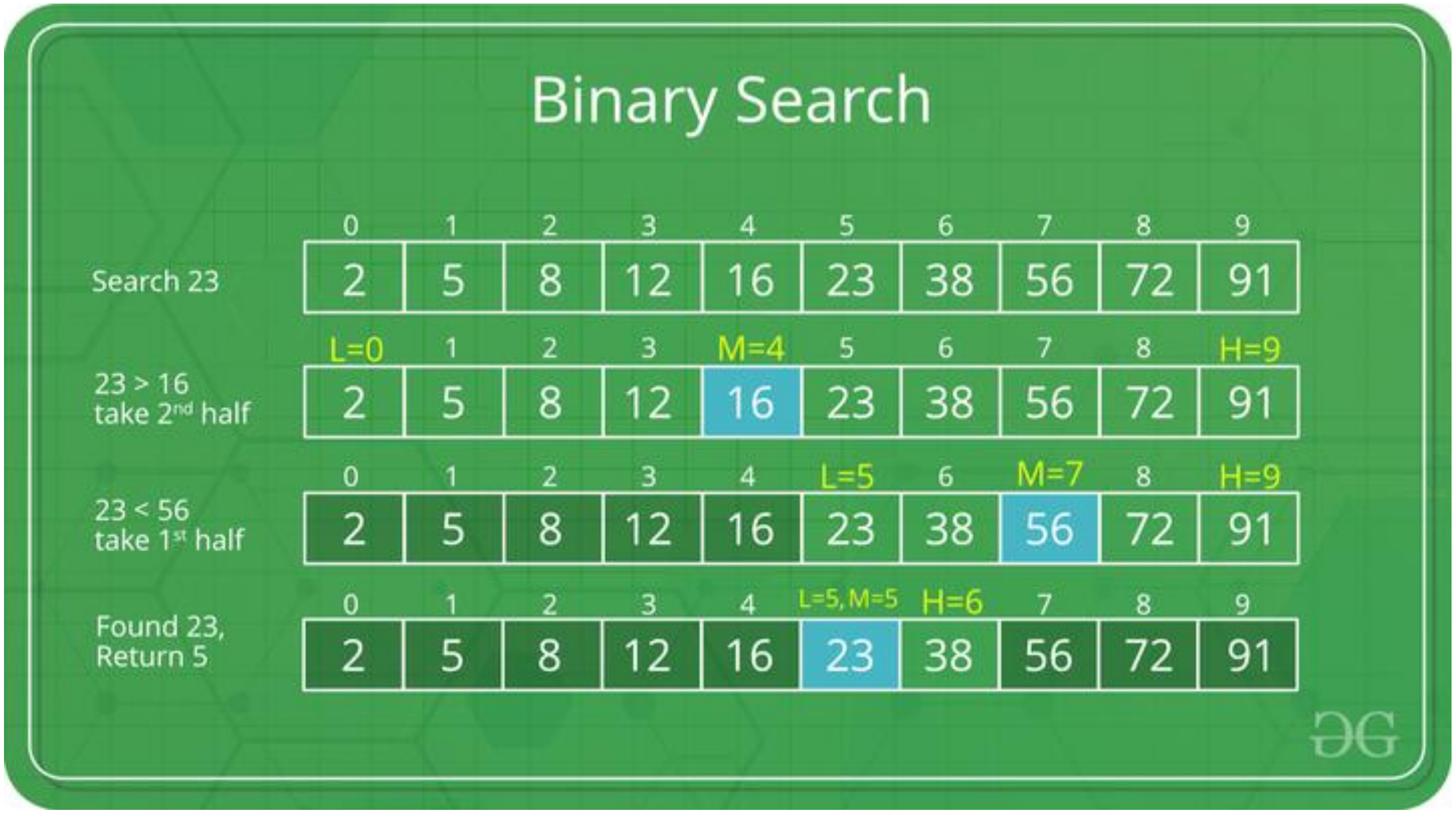
4.1. Introduction

- Linear Search to find the element "20" in a given list of numbers.



4.1. Introduction

- Binary Search to find the element "23" in a given list of numbers.



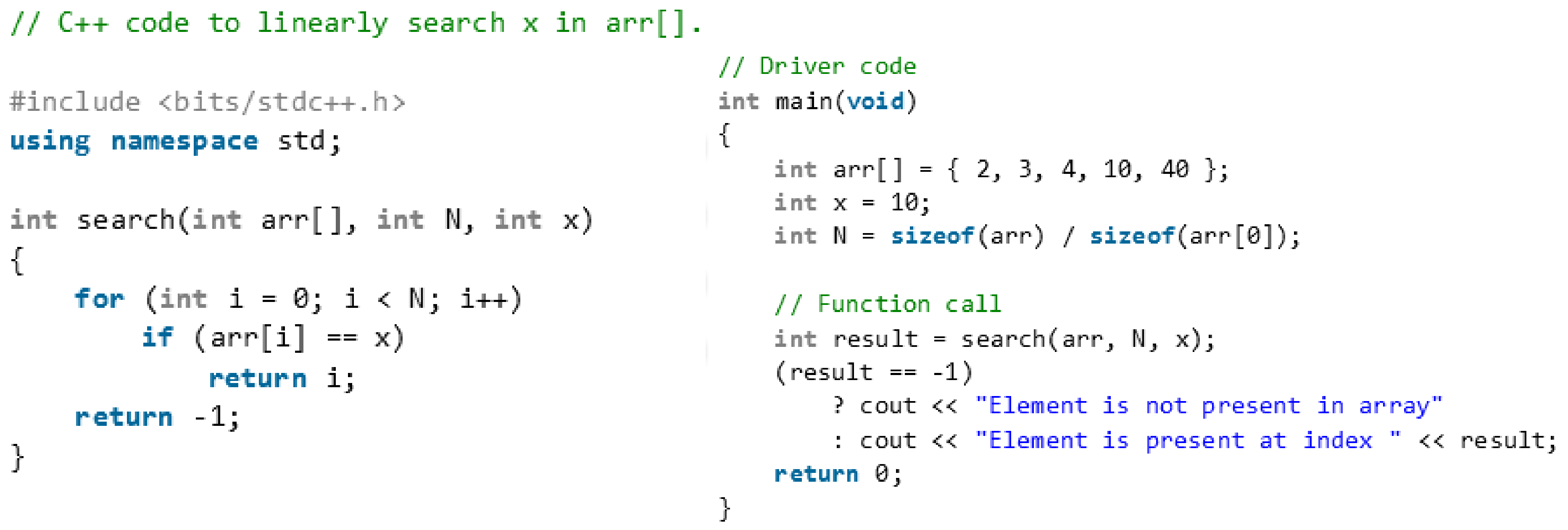
	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91

4.2. Linear Search

- **Linear Search** is defined as a *sequential search algorithm* that starts at one end, and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.
- In Linear Search Algorithm:
  - Every element is considered as a potential match for the key and checked for the same.
  - If any element is found equal to the key, the search is successful and the index of that element is returned.
  - If no element is found equal to the key, the search yields "No match found".

4.2. Linear Search

- Below is the implementation of the linear search algorithm:



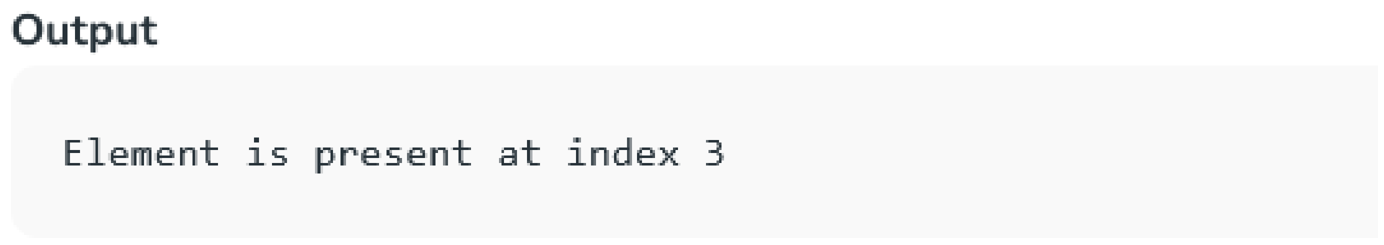
```cpp
// C++ code to linearly search x in arr[].

#include <bits/stdc++.h>
using namespace std;

int search(int arr[], int N, int x)
{
    for (int i = 0; i < N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    int result = search(arr, N, 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
```

4.2. Linear Search

- **Time Complexity:**
  - **Best Case:** In the best case, the key might be present at the first index. So the best case complexity is O(1)
  - **Worst Case:** In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.
  - **Average Case:** O(N)
- **Auxiliary Space:** O(1) as except for the variable to iterate through the list, no other variable is used.

4.2. Linear Search

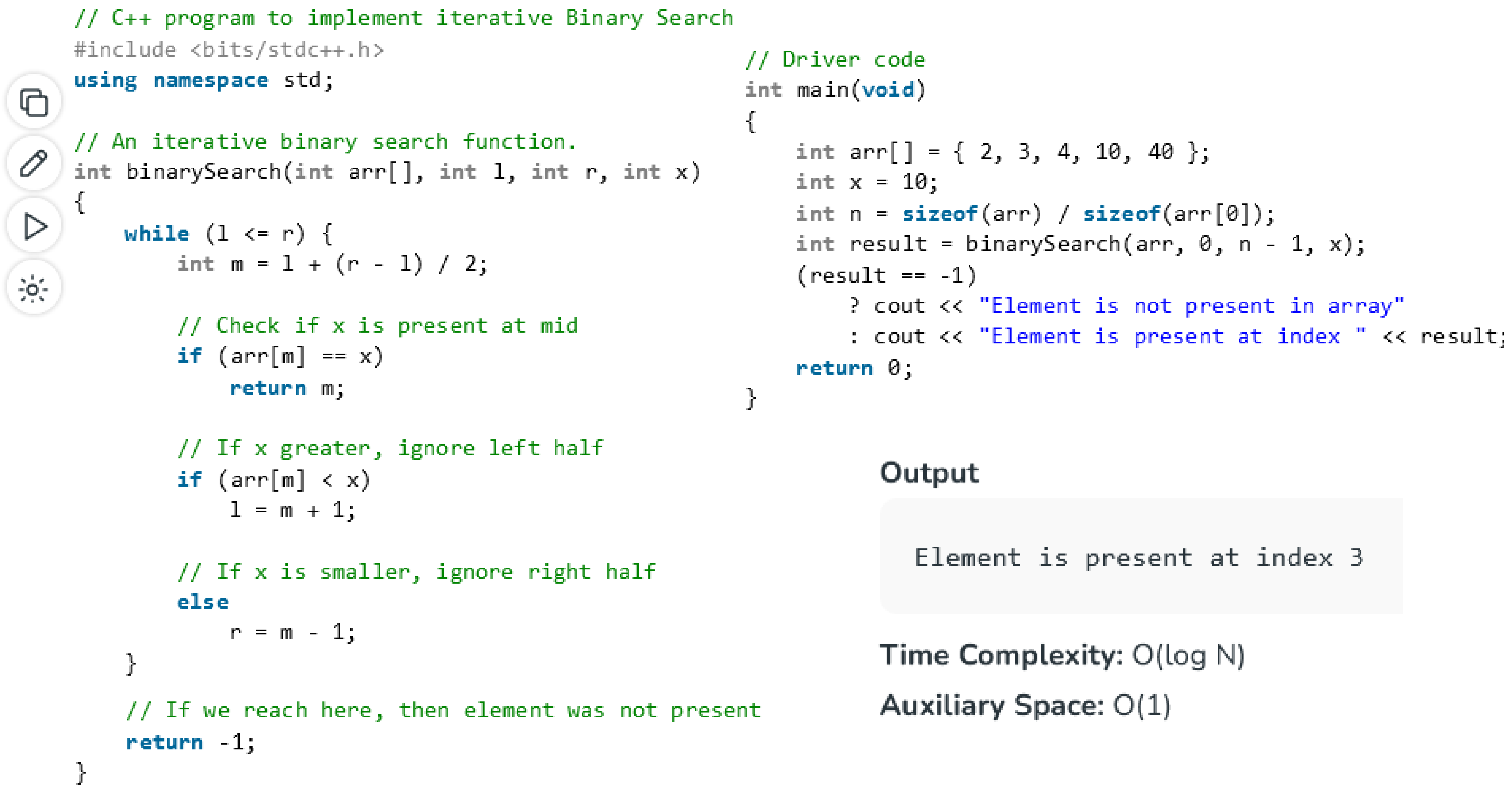
- Linear search has several practical applications in computer science and beyond. Some examples:
  - **Phonebook Search**: Linear search can be used to search through a phonebook to find a person’s name, given their phone number.
  - **Spell Checkers**: The algorithm compares each word in the document to a dictionary of correctly spelled words until a match is found.
  - **Finding Minimum and Maximum Values:** Linear search can be used to find the minimum and maximum values in an array or list.
  - **Searching through unsorted data:** Linear search is useful for searching through unsorted data.

4.3. Binary Search

- **Binary Search** is defined as a searching algorithm performed on a *sorted* data structure, e.g. array, by ***repeatedly dividing the search interval in half.***
- Searching is done by dividing the array into two halves. It utilizes the ***divide-and-conquer approach*** to find an element. To apply Binary Search:
  - The data structure must be sorted.
  - Access to any element of the data structure takes constant time.

4.3. Binary Search

- Below is the implementation of the binary search algorithm:



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

// An iterative binary search function.
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;
}
```

```cpp
// Driver code
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)

4.3. Binary Search

- **Time Complexity:**
  - Best Case: O(1)
  - Average Case: O(log N)
  - Worst Case: O(log N)
- **Auxiliary Space:** O(1).
- ***Binary search*** is *faster* than ***linear search***, especially for large arrays. More efficient than other searching algorithms with a similar time complexity, such as ***interpolation search*** or ***exponential search.***
- 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.

4.3. Binary Search

- 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.
  - It can be used for searching in ***computer graphics***, such as algorithms for ray tracing or texture mapping.
  - It can be used for ***searching a database***.
  - In real life, binary search can be applied in the ***dictionary***.
  - Binary search is also used to ***debug*** a linear piece of code.
  - Binary search is also used to find if a number is a ***square*** of another or not.

4.3. Binary Search

- We are given a sorted array A[ ] of n elements. We need to find if x is present in A or not.
- In Binary Search we always used middle element, but in Randomized Binary Search we will randomly pick one element in given range.
- In Binary search we had:

$$\text{middle} = (\text{start} + \text{end})/2$$

4.3. Binary Search

- In **Randomized Binary Search** we do following:
  - Generate a random number t;
  - Since range of number in which we want a random number is [start, end];
  - Hence we do, t = t % (end-start + 1);
  - Then, t = start + t;
  - Hence t is a random number between start and end.
- ***Expected Time complexity*** of Randomized Binary Search Algorithm for n elements is: T(n) = O(log n).

4.3. Binary Search

- Implementation of randomized binary search algorithm:

```cpp
// C++ program to implement iterative
// randomized algorithm.
#include <iostream>
#include <ctime>
using namespace std;

// To generate random number
// between x and y ie.. [x, y]
int getRandom(int x, int y)
{
    srand(time(NULL));
    return (x + rand()%(y-x+1));
}

// A iterative randomized binary search function.
// It returns location of x in
// given array arr[l..r] if present, otherwise -1
int randomizedBinarySearch(int arr[], int l,
                                int r, int x)
{
    while (l <= r)
    {
        // Here we have defined middle as
        // random index between l and r ie.. [l, r]
        int m = getRandom(l, r);

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

4.3. Binary Search

- **Binary search** and **Ternary search**
- Which of the above two does *less* comparisons in worst case?
- From the first look at the *recursive functions* of both algorithms, it seems the ternary search does less number of comparisons, as it makes $\text{Log}_3(n)$ recursive calls, but binary search makes $\text{Log}_2(n)$ recursive calls.
- Let us take a closer look.

4.3. Binary Search

- The following is recursive formula for counting comparisons in worst case of Binary Search:

$$T(n) = T(n/2) + 2, T(1) = 1.$$

- The following is recursive formula for counting comparisons in worst case of Ternary Search:

$$T(n) = T(n/3) + 4, T(1) = 1$$

- In Binary search, there are:

  $2*\mathrm{Log}_2(n) + 1$ comparisons in worst case.

- In Ternary search, there are:

  $4*\mathrm{Log}_3(n) + 1$ comparisons in worst case.

4.3. Binary Search

- Time Complexity for Binary search

$$= 2*c*Log_2(n) \ + O(1)$$

- Time Complexity for Ternary search

$$= 4*c*Log_3(n) \ + O(1)$$

- Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions: $2*Log_3(n)$ and $Log_2(n)$.
- The value of $2*Log_3(n)$ is equal to $(2/Log_2 3)*Log_2(n)$.
- Since the value of $(2/Log_2 3) > 1$, Ternary Search does *more* comparisons than Binary Search in worst case.

4.3. Binary Search

- **Binary Search** is a search algorithm that is specifically designed for searching in **sorted** data structures. This algorithm is much more efficient than **Linear Search.**
- Now, the question arises, is Binary Search applicable to unsorted arrays?
- So, the answer is **no,** it is not possible to use or implement Binary Search on unsorted arrays or lists, because, the *repeated targeting* of the *mid element* of one half *depends on the sorted order* of data structure.

4.4. Jump Search

- Like *Binary Search*, **Jump Search** is a searching algorithm for sorted arrays.
- The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.
- For example, suppose we have an array arr[ ] of size n and a block (to be jumped) of size m. Then we search in the indexes arr[0], arr[m], arr[2m],…, arr[km], and so on. Once we find the interval (arr[km] $<$ x $<$ arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

4.4. Jump Search

- Performance of *jump search* in comparison to *linear* and *binary search*:
  - If we compare it with linear and binary search then it comes out then it is *better* than linear search, but *not better* than binary search.
- The increasing order of performance is:

  linear search < *jump search* < *binary search*
- What is the optimal block size to be skipped?
  - The best step size is **m = √n.**

4.4. Jump Search

- Implementation of jump search algorithm:

```cpp
// C++ program to implement Jump Search

#include <bits/stdc++.h>
using namespace std;

int jumpSearch(int arr[], int x, int n)
{
    // Finding block size to be jumped
    int step = sqrt(n);

    // Finding the block where element is
    // present (if it is present)
    int prev = 0;
    while (arr[min(step, n)-1] < x)
    {
        prev = step;
        step += sqrt(n);
        if (prev >= n)
            return -1;
    }

    // Doing a linear search for x in block
    // beginning with prev.
    while (arr[prev] < x)
    {
        prev++;

        // If we reached next block or end of
        // array, element is not present.
        if (prev == min(step, n))
            return -1;
    }
    // If element is found
    if (arr[prev] == x)
        return prev;

    return -1;
}

// Driver program to test function
int main()
{
    int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,
                34, 55, 89, 144, 233, 377, 610 };
    int x = 55;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Find the index of 'x' using Jump Search
    int index = jumpSearch(arr, x, n);

    // Print the index where 'x' is located
    cout << "\nNumber " << x << " is at index " << index;
    return 0;
}
```

Output:

```
Number 55 is at index 10
```

Time Complexity : O(√n)

Auxiliary Space : O(1)

4.4. Jump Search

- **Advantages of Jump Search:**
  - Better than a linear search for arrays where the elements are uniformly distributed.
  - Jump search has a lower time complexity compared to a linear search for large arrays.
  - The number of steps taken in jump search is *proportional* to the ***square root of the size of the array***, making it more efficient for large arrays.
  - It is easier to implement compared to other search algorithms, like binary search or ternary search.
  - It works well where the elements are uniformly distributed, as it can jump to a closer position with each iteration.

4.4. Jump Search

- **Important points:**
  - Works only with sorted arrays.
  - The optimal size of a block to be jumped is: **√n.** This makes the time complexity of Jump Search to be **O(√n)**.
  - The time complexity of Jump Search is between Linear Search, **O(n)**, and Binary Search, **O(Log n)**.
  - Binary Search is *better* than Jump Search, but Jump Search has the advantage that we traverse back *only once* (Binary Search may require up to O(Log n) jumps, consider a situation where the element to be searched is the smallest element or just bigger than the smallest). So, in a system where Binary search is costly, we use Jump Search.