8.2 Searching - Linear vs Binary

Wednesday, June 9, 2021 1:55 AM

Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

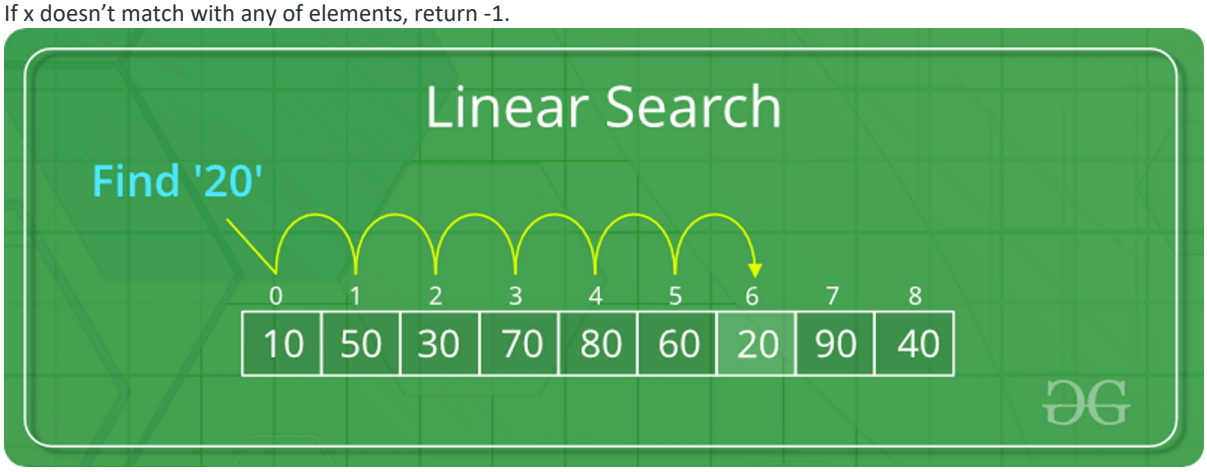
Examples :

```
Input : arr[] = {10, 20, 80, 30, 60, 50,
                 110, 100, 130, 170}
          x = 110;
Output : 6
Element x is present at index 6
Input : arr[] = {10, 20, 80, 30, 60, 50,
                 110, 100, 130, 170}
           x = 175;
Output : -1
Element x is not present in arr[].
```

Recommended: Please solve it on "***PRACTICE***" first, before moving on to the solution.

A simple approach is to do a linear search, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.



Example:

C++

```
// C++ code to linearly search x in arr[]. If x
// is present then return its location, otherwise
// return -1

#include <iostream>
using namespace std;

int search(int arr[], int n, int x)
{
    int i;
    for (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;
}
```

The **time complexity** of the above algorithm is O(n).

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster-searching comparison to Linear search.

Improve Linear Search Worst-Case Complexity

1. if element Found at last O(n) to O(1)
2. if element Not found O(n) to O(n/2)
   Below is the implementation:

C++14

```
// C++ program for linear search
#include<bits/stdc++.h>
using namespace std;

void search(vector<int> arr, int search_Element)
{
    int left = 0;
    int length = arr.size();
    int position = -1;
      int right = length - 1;

    // Run loop from 0 to right
    for(left = 0; left <= right;)
    {

        // If search_element is found with
        // left variable
        if (arr[left] == search_Element)
        {

            position = left;
            cout << "Element found in Array at "
                 << position + 1 << " Position with "
                 << left + 1 << " Attempt";

            break;
        }

        // If search_element is found with
        // right variable
        if (arr[right] == search_Element)
        {
            position = right;
            cout << "Element found in Array at "
                 << position + 1 << " Position with "
                 << length - right << " Attempt";

            break;
        }
        left++;
        right--;
    }

    // If element not found
    if (position == -1)
        cout << "Not found in Array with "
             << left << " Attempt";
}

// Driver code
int main()
{
    vector<int> arr{ 1, 2, 3, 4, 5 };
    int search_element = 5;

    // Function call
    search(arr, search_element);
}
```

```
// This code is contributed by mayanktyagi1709
```

From <https://www.geeksforgeeks.org/linear-search/>

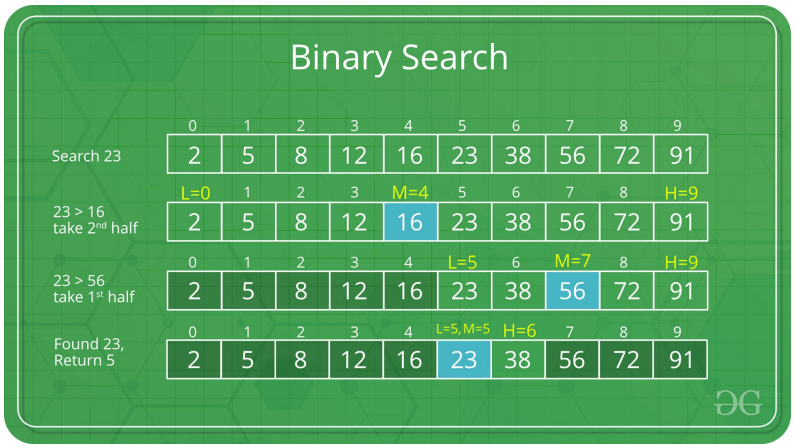
Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do a **linear search.** The time complexity of the above algorithm is O(n). Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

Recommended: Please solve it on "***PRACTICE***" first, before moving on to the solution.

We basically ignore half of the elements just after one comparison.

3. Compare x with the middle element.
4. If x matches with the middle element, we return the mid index.
5. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
6. Else (x is smaller) recur for the left half.

Recursive implementation of Binary Search

Recursive implementation of Binary Search

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

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
```

```
        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

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
```

Here you can create a check function for easier implementation.

Here is recursive implementation with check function which I feel is a much easier implementation:

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

//define array globally
const int N = 1e6 +4;

int a[N];
int n;//array size

//elememt to be searched in array
   int k;

bool check(int dig)
{
    //element at dig position in array
    int ele=a[dig];

    //if k is less than
    //element at dig position
    //then we need to bring our higher ending to dig
    //and then continue further
    if(k<=ele)
    {
        return 1;
    }
    else
    {
    return 0;
    }
}
void binsrch(int lo,int hi)
{
    while(lo<hi)
    {
        int mid=(lo+hi)/2;
        if(check(mid))
        {
            hi=mid;
        }
        else
        {
            lo=mid+1;
        }
    }
    //if a[lo] is k
    if(a[lo]==k)
        cout<<"Element found at index "<<lo;// 0 based indexing
    else
        cout<<"Element doesnt exist in array";//element was not in our
array

}
```

```
int main()
{
    cin>>n;
   for(int i=0; i<n; i++)
   {
    cin>>a[i];
   }
   cin>>k;

   //it is being given array is sorted
   //if not then we have to sort it

   //minimum possible point where our k can be is starting index
   //so lo=0
   //also k cannot be outside of array so end point
   //hi=n

   binsrch(0,n);

    return 0;
}
```

Iterative implementation of Binary Search

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

// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
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;
}

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:

The time complexity of Binary Search can be written as

```
T(n) = T(n/2) + c
```

The above recurrence can be solved either using the Recurrence Tree method or Master method. It falls in case II of the Master Method and the solution of the recurrence is .

Auxiliary Space: O(1) in case of iterative implementation. In the case of recursive implementation, O(Logn) recursion call stack space.

Algorithmic Paradigm: Decrease and Conquer.

Note:

Here we are using

int mid = low + (high – low)/2;

Maybe, you wonder why we are calculating the ***middle index*** this way, we can simply add the *lower and higher index and divide it by 2.*

int mid = (low + high)/2;

But if we calculate the ***middle index*** like this means our code is not 100% correct, it contains bugs.

That is, it fails for larger values of int variables low and high. Specifically, it fails if the sum of low and high is greater than the maximum positive int value($2_{31} – 1$).

The sum overflows to a negative value and the value stays negative when divided by 2. In java, it throws *ArrayIndexOutOfBoundException.*

int mid = low + (high – low)/2;

So it's better to use it like this. This bug applies equally to merge sort and other divide and conquer algorithms.

From <https://www.geeksforgeeks.org/binary-search/>

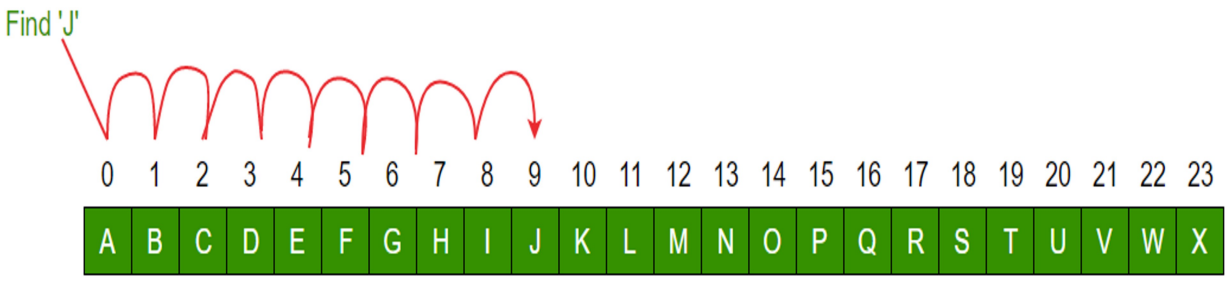
Linear Search vs Binary Search

7. The worst case complexity is O(n), sometimes known an O(n) search
8. Time taken to search elements keep increasing as the number of elements are increased.
   A binary search however, cut down your search to half as soon as you find middle of a sorted list.
9. The middle element is looked to check if it is greater than or less than the value to be searched.
10. Accordingly, search is done to either half of the given list
    Important Differences

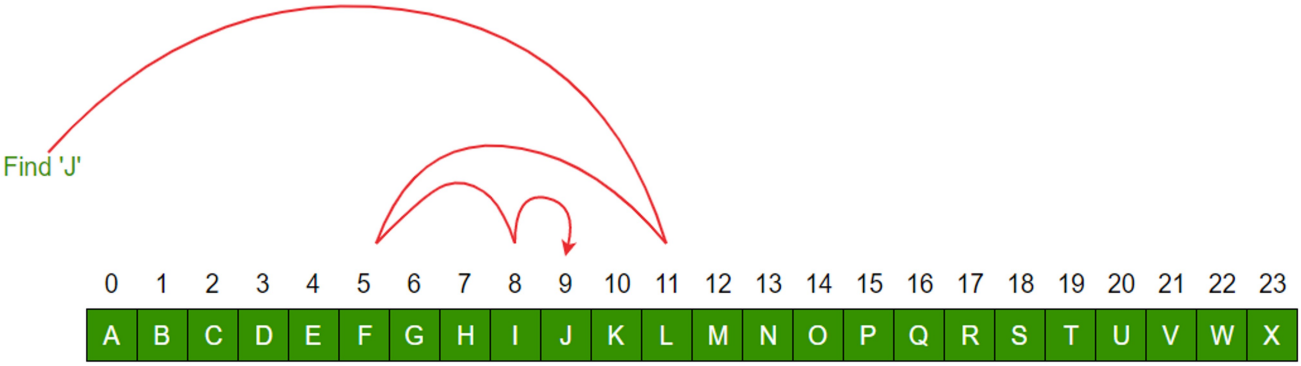
- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search -O(n) , Binary search has time complexity O(log n).
- Linear search performs equality comparisons and Binary search performs ordering comparisons

Let us look at an example to compare the two:

Linear Search to find the element "J" in a given sorted list from A-X



Binary Search to find the element "J" in a given sorted list from A-X



You may also see

- Searching and Sorting Articles
- Searching and Sorting Quizzes
- Practice Coding Questions

From <https://www.geeksforgeeks.org/linear-search-vs-binary-search/>