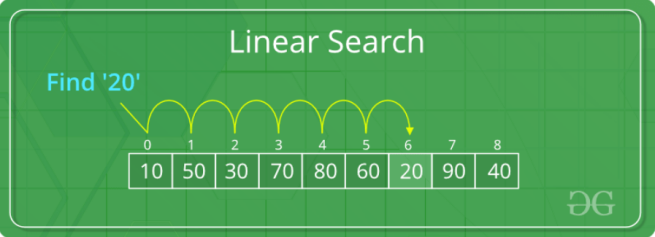
# **Linear Search Algorithm** :

# **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.



**How Does Linear Search Algorithm Work?**

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".

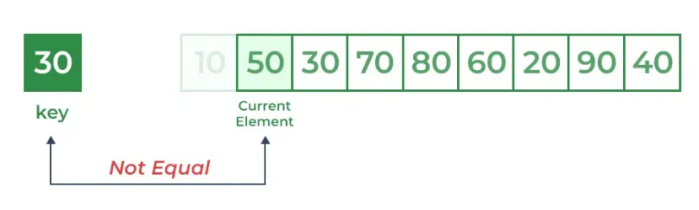
For example: Consider the array arro = {10, 50, 30, 70, 80, 20, 90, 40} and key = 30

***Step 1:****Start from the first element (index 0) and compare****key****with each element (arr[i]).*

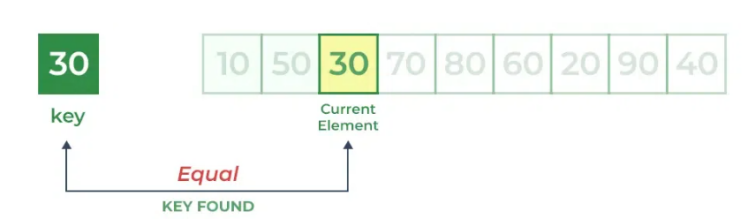
• Comparing key with first element arr[O]. Since not equal, the iterator moves to the next element as a potential match.



• Comparing key with next element arr[l]. Since not equal, the iterator moves to the next element as a potential match.



***Step 1:****Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).*



#include <stdio.h>

int search(int arr[], int N, int x)

{

for (int i = 0; i < N; i++)

if (arr[i] == x)

return i;

return -1;

}

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)

? printf("Element is not present in array")

: printf("Element is present at index %d", result);

return 0;

}

## **Application of Linear search algorithm :**

Finding an Item in a List or Array:

Linear search is commonly used to locate an element in an unordered list or array.

*Phonebook Search:*

When searching for a contact in a phonebook or a contact list on a mobile phone, a linear search may be employed.

*Student Grade Lookup:*

Linear search can be used to find the grades of a particular student in a class or school database.

*To-Do List Management:*

Linear search can help in managing and searching through a to-do list to find specific tasks.

*Library Book Lookup:*

In a library catalog, linear search can be used to find the location or availability status of a particular book.

*Checking for Duplicates:*

Linear search can identify duplicate elements in a list or array, which is useful in data cleaning and validation processes.

*Employee Database Search:*

In human resources, linear search can be used to look up employee information based on employee IDs or names.

*Search in a File System:*

Linear search can be applied to find the location of a file in a directory or file system.

*Password Validation:*

Linear search can be used to check whether a given password matches any of the stored passwords in a system.

*Checking Membership:*

Linear search can determine if a particular element is a member of a set or group.

*Searching for a Song in a Playlist:*

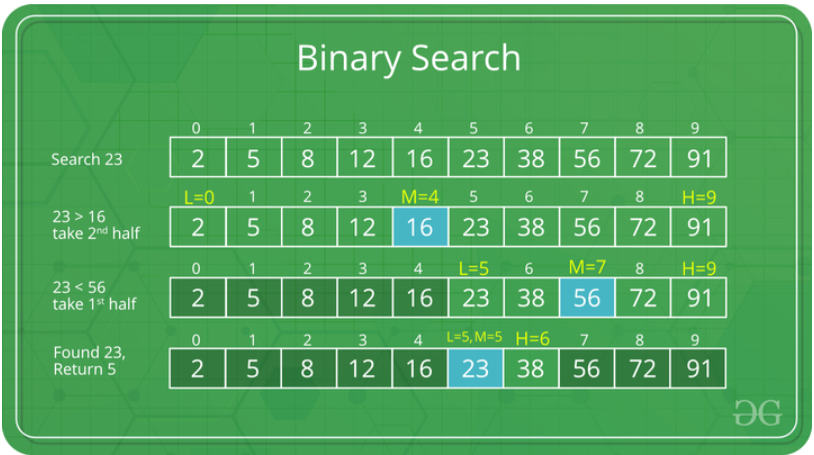
Finding a particular song in a playlist on a music streaming app.

*Finding a TV Channel:*

Scrolling through channels on your TV to find a specific one.

# **Binary Search Algorithm** :

Binary Search is defined as a [searching algorithm](https://www.geeksforgeeks.org/searching-algorithms/) used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).



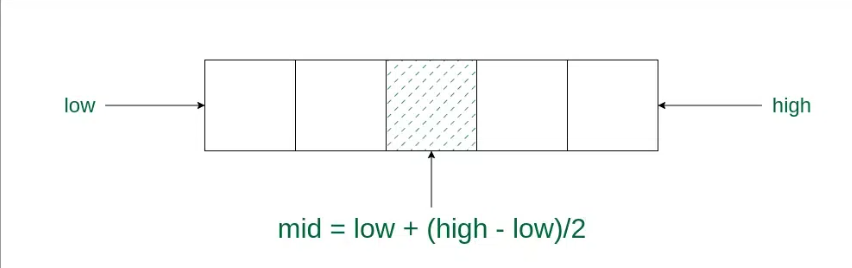
**Conditions for when to apply Binary Search in a Data Structure:**

* The data structure must be sorted.
* Access to any element of the data structure takes constant time.

## **Binary Search Algorithm:**

in this algorithm,

* Divide the search space into two halves by [finding the middle index “mid”](https://www.geeksforgeeks.org/problem-binary-search-implementations/).



* Compare the middle element of the search space with the key.
* If the key is found at middle element, the process is terminated.
* If the key is not found at middle element, choose which half will be used as the next search

space.

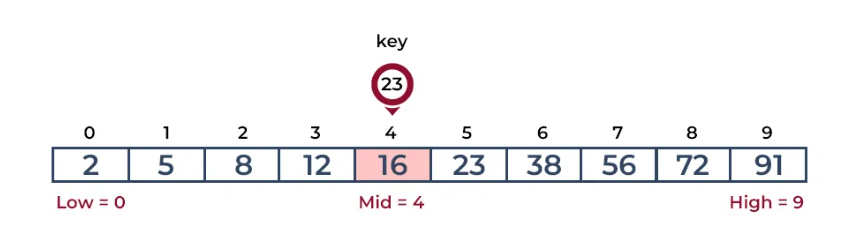
* *If the key is smaller than the middle element, then the left side is used for next search.*
* *If the key is larger than the middle element, then the right side is used for next search.*
* This process is continued until the key is found or the total search space is exhausted.

## How does Binary Search work?

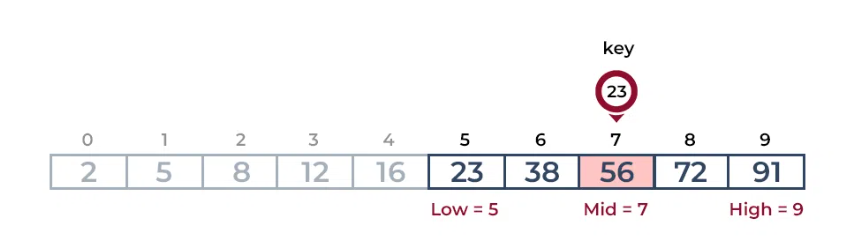
Consider an array **arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}**, and the **target = 23**.

**First Step:** Calculate the mid and compare the mid element with the key. If the key is less than mid element, move to left and if it is greater than the mid then move search space to the right.

* Key (i.e., 23) is greater than current mid element (i.e., 16). The search space moves to the right.

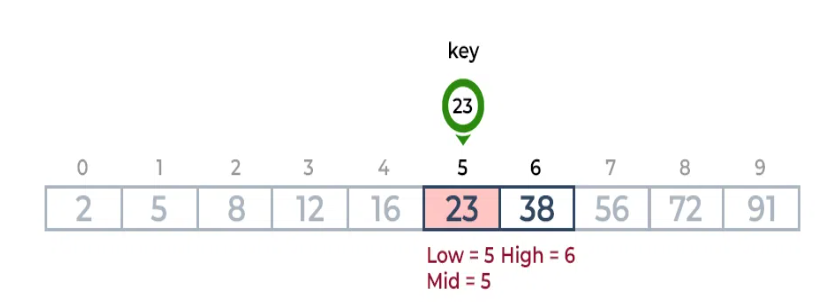


* Key is less than the current mid 56. The search space moves to the left.x



* Second Step: If the key matches the value of the mid element, the element is found and stop

search.



## How to Implement Binary Search?

The Binary Search Algorithm can be implemented in the following two ways

1. Iterative Binary Search Algorithm
2. Recursive Binary Search Algorithm

## **Iterative  Binary Search Algorithm:**

Here we use a while loop to continue the process of comparing the key and splitting the search space in two halves.

// C program to implement iterative Binary Search

#include <stdio.h>

// 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;

}

int main(void)

{

int arr[] = { 2, 3, 4, 10, 40 };

int n = sizeof(arr) / sizeof(arr[0]);

int x = 10;

int result = binarySearch(arr, 0, n - 1, x);

(result == -1) ? printf("Element is not present in array"): printf("Element is present at index %d", result);

return 0;

}

## **Recursive  Binary Search Algorithm:**

Create a recursive function and compare the mid of the search space with the key. And based on the result either return the index where the key is found or call the recursive function for the next search space.

// C program to implement recursive Binary Search

#include <stdio.h>

// 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()

{

int arr[] = { 2, 3, 4, 10, 40 };

int n = sizeof(arr) / sizeof(arr[0]);

int x = 10;

int result = binarySearch(arr, 0, n - 1, x);

(result == -1)

? printf("Element is not present in array")

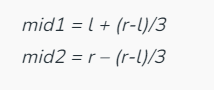
: printf("Element is present at index %d", result);

return 0;

}

# **Ternary Search Algorithm** :

Ternary search is a searching algorithm that is used to find the position of a target value within a sorted array. It operates on the principle of dividing the array into three parts instead of two, as in [binary search](https://www.geeksforgeeks.org/binary-search/). The basic idea is to narrow down the search space by comparing the target value with elements at two points that divide the array into three equal parts.



## **Working of Ternary Search:**

The concept involves dividing the array into three equal segments and determining in which segment the key element (the element being sought) is located. It works similarly to a binary search, with the distinction of reducing time complexity by dividing the array into three parts instead of two.

Below are the step-by-step explanation of working of Ternary Search:

1. Initialization:

* Begin with a sorted array.
* Set two pointers, left and right, initially pointing to the first and last elements of the array.

1. Divide the Array:

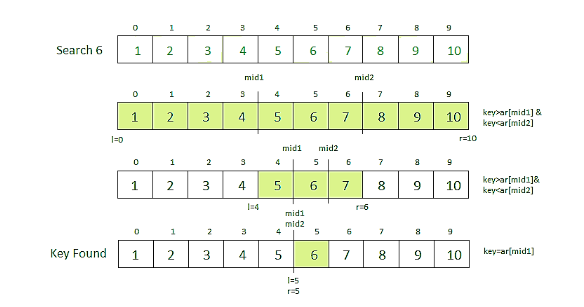
* Calculate two midpoints, mid1 and mid2, dividing the current search space into three roughly equal parts:
* mid1 = left + (right – left) / 3
* mid2 = right – (right – left) / 3
* The array is now effectively divided into [left, mid1], (mid1, mid2), and [mid2, right].

1. Comparison with Target:

* If the target is equal to the element at mid1 or mid2, the search is successful, and the index is returned
* If the target is less than the element at mid1, update the right pointer to mid1 – 1.
* If the target is greater than the element at mid2, update the left pointer to mid2 + 1.
* If the target is between the elements at mid1 and mid2, update the left pointer to mid1 + 1 and the right pointer to mid2 – 1.

1. Repeat or Conclude:

* Repeat the process with the reduced search space until the target is found or the search space becomes empty.
* If the search space is empty and the target is not found, return a value indicating that the target is not present in the array.



1. Iterative Implementation of Ternary Search:

// C program to illustrate

// iterative approach to ternary search

#include <stdio.h>

// Function to perform Ternary Search

int ternarySearch(int l, int r, int key, int ar[])

{

while (r >= l) {

// Find the mid1 and mid2

int mid1 = l + (r - l) / 3;

int mid2 = r - (r - l) / 3;

// Check if key is present at any mid

if (ar[mid1] == key) {

return mid1;

}

if (ar[mid2] == key) {

return mid2;

}

// Since key is not present at mid,

// check in which region it is present

// then repeat the Search operation

// in that region

if (key < ar[mid1]) {

// The key lies in between l and mid1

r = mid1 - 1;

}

else if (key > ar[mid2]) {

// The key lies in between mid2 and r

l = mid2 + 1;

}

else {

// The key lies in between mid1 and mid2

l = mid1 + 1;

r = mid2 - 1;

}

}

// Key not found

return -1;

}

// Driver code

int main()

{

int l, r, p, key;

// Get the array

// Sort the array if not sorted

int ar[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Starting index

l = 0;

// end element index

r = 9;

// Checking for 5

// Key to be searched in the array

key = 5;

// Search the key using ternarySearch

p = ternarySearch(l, r, key, ar);

// Print the result

printf("Index of %d is %d\n", key, p);

// Checking for 50

// Key to be searched in the array

key = 50;

// Search the key using ternarySearch

p = ternarySearch(l, r, key, ar);

// Print the result

printf("Index of %d is %d", key, p);

}

2. Recursive Implementation of Ternary Search:

// C program to illustrate

// recursive approach to ternary search

#include <stdio.h>

// Function to perform Ternary Search

int ternarySearch(int l, int r, int key, int ar[])

{

if (r >= l) {

// Find the mid1 and mid2

int mid1 = l + (r - l) / 3;

int mid2 = r - (r - l) / 3;

// Check if key is present at any mid

if (ar[mid1] == key) {

return mid1;

}

if (ar[mid2] == key) {

return mid2;

}

// Since key is not present at mid,

// check in which region it is present

// then repeat the Search operation

// in that region

if (key < ar[mid1]) {

// The key lies in between l and mid1

return ternarySearch(l, mid1 - 1, key, ar);

}

else if (key > ar[mid2]) {

// The key lies in between mid2 and r

return ternarySearch(mid2 + 1, r, key, ar);

}

else {

// The key lies in between mid1 and mid2

return ternarySearch(mid1 + 1, mid2 - 1, key, ar);

}

}

// Key not found

return -1;

}

// Driver code

int main()

{

int l, r, p, key;

// Get the array

// Sort the array if not sorted

int ar[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Starting index

l = 0;

// end element index

r = 9;

// Checking for 5

// Key to be searched in the array

key = 5;

// Search the key using ternarySearch

p = ternarySearch(l, r, key, ar);

// Print the result

printf("Index of %d is %d\n", key, p);

// Checking for 50

// Key to be searched in the array

key = 50;

// Search the key using ternarySearch

p = ternarySearch(l, r, key, ar);

// Print the result

printf("Index of %d is %d", key, p);

}

## **Advantages:**

Ternary search can find maxima/minima for unimodal functions, where binary search is not applicable.

Ternary Search has a time complexity of O(2 \* log3n), which is more efficient than linear search and comparable to binary search.

Fits well with optimization problems.

## **Disadvantages:**

When you have a large ordered array or list and need to find the position of a specific value.

When you need to find the maximum or minimum value of a function.

When you need to find bitonic point in a [bitonic](https://www.geeksforgeeks.org/bitonic-sort/)sequence.

When you have to evaluate a quadratic expression

# **Jump Search Algorithm:**

Like [Binary Search](https://www.geeksforgeeks.org/binary-search/), Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than [linear search](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/)) 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.  
Let’s consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). The length of the array is 16. The Jump search will find the value of 55 with the following steps assuming that the block size to be jumped is 4.   
STEP 1: Jump from index 0 to index 4;   
STEP 2: Jump from index 4 to index 8;   
STEP 3: Jump from index 8 to index 12;   
STEP 4: Since the element at index 12 is greater than 55, we will jump back a step to come to index 8.   
STEP 5: Perform a linear search from index 8 to get the element 55.

#### Performance 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

Implementation of JUmp Search:

#include<stdio.h>

#include<math.h>

int min(int a, int b){

if(b>a)

return a;

else

return b;

}

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;

}

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]);

int index = jumpsearch(arr, x, n);

if(index >= 0)

printf("Number is at %d index",index);

else

printf("Number is not exist in the array");

return 0;

}

#### **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.

Jump search works well for arrays where the elements are in order and uniformly distributed, as it can jump to a closer position in the array with each iteration.