# **Binary Search**

**(Бінарний пошук)**

Алгоритм бінарного пошуку: Основні кроки для виконання бінарного пошуку такі:

* Відсортуйте масив за зростанням.
* Першому елементу масиву присвоїти молодший індекс, а останньому - старший.
* Середній індекс дорівнює середньому значенню нижнього та верхнього індексів.
* Якщо елемент з середнім індексом є цільовим елементом, повернути середній індекс.
* Якщо цільовий елемент менший за елемент з середнім індексом, встановіть верхній індекс до середнього індексу - 1.
* Якщо цільовий елемент більший за елемент з середнім індексом, встановіть нижній індекс рівним середньому індексу + 1.
* Повторюйте кроки 3-6, поки елемент не буде знайдено або поки не стане зрозуміло, що його немає в масиві.

Алгоритм бінарного пошуку може бути реалізований наступними двома способами:

* Ітеративний метод (**Iterative Method**)
* Рекурсивний метод (**Recursive Method**)

1. Iteration Method

binarySearch(arr, x, low, high)

repeat till low = high

mid = (low + high)/2

if (x == arr[mid])

return mid

else if (x > arr[mid]) // x is on the right side

low = mid + 1

else // x is on the left side

high = mid - 1

2. Recursive Method (The recursive method follows the divide and conquer approach)

binarySearch(arr, x, low, high)

if low > high

return False

else

mid = (low + high) / 2

if x == arr[mid]

return mid

else if x > arr[mid] // x is on the right side

return binarySearch(arr, x, mid + 1, high)

else // x is on the left side

return binarySearch(arr, x, low, mid - 1)

**Покроковий алгоритм бінарного пошуку (Step-by-step Binary Search Algorithm):** Ми ігноруємо половину елементів після одного порівняння.

* Порівнюємо x з середнім елементом.
* Якщо x збігається з середнім елементом, ми повертаємо середній індекс.
* Інакше, якщо x більший за середній елемент, то x може знаходитись лише у правій половині підмасиву після середнього елемента. Тому ми повторюємо для правої половини.
* Інакше (x менше) повторюємо для лівої половини.

**Recursive implementation of Binary Search**:

// C# implementation of recursive Binary Search

using System;

class GFG {

// Returns index of x if it is present in

// arr[l..r], else return -1

static 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;

}

// Driver method to test above

public static void Main()

{

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

int n = arr.Length;

int x = 10;

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

if (result == -1)

Console.WriteLine("Element not present");

else

Console.WriteLine("Element found at index "

+ result);

}

}

**Another Iterative Approach to Binary Search**

using System;

public class GFG{

static public void binarySearch(int[] v, int To\_Find)

{

int lo = 0;

int hi = v.Length - 1;

int mid;

// This below check covers all cases , so need to check

// for mid=lo-(hi-lo)/2

while (hi - lo > 1) {

mid = (hi + lo) / 2;

if (v[mid] < To\_Find) {

lo = mid + 1;

}

else {

hi = mid;

}

}

if (v[lo] == To\_Find) {

Console.WriteLine("Found At Index " + lo);

}

else if (v[hi] == To\_Find) {

Console.WriteLine("Found At Index " + hi);

}

else {

Console.WriteLine("Not Found");

}

}

static public void Main (){

int[] v = { 1, 3, 4, 5, 6 };

int To\_Find = 1;

binarySearch(v, To\_Find);

To\_Find = 6;

binarySearch(v, To\_Find);

To\_Find = 10;

binarySearch(v, To\_Find);

}

}

**Iterative implementation of Binary Search**

// C# implementation of iterative Binary Search

using System;

class GFG {

// Returns index of x if it is present in arr[],

// else return -1

static int binarySearch(int[] arr, int x)

{

int l = 0, r = arr.Length - 1;

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;

}

// Driver method to test above

public static void Main()

{

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

int n = arr.Length;

int x = 10;

int result = binarySearch(arr, x);

if (result == -1)

Console.WriteLine("Element not present");

else

Console.WriteLine("Element found at "

+ "index " + result);

}

}

**Переваги бінарного пошуку:**

* Бінарний пошук швидший за лінійний, особливо для великих масивів. Зі збільшенням розміру масиву час, необхідний для виконання лінійного пошуку, зростає лінійно, тоді як час, необхідний для виконання бінарного пошуку, зростає логарифмічно.
* Бінарний пошук є більш ефективним, ніж інші алгоритми пошуку, які мають подібну часову складність, наприклад, інтерполяційний пошук або експоненціальний пошук.
* Бінарний пошук відносно простий у реалізації і зрозумілий, що робить його гарним вибором для багатьох застосувань.
* Бінарний пошук можна використовувати як для відсортованих масивів, так і для відсортованих зв'язаних списків, що робить його гнучким алгоритмом.
* Бінарний пошук добре підходить для пошуку великих наборів даних, які зберігаються у зовнішній пам'яті, наприклад, на жорсткому диску або в хмарі.
* Бінарний пошук можна використовувати як будівельний блок для більш складних алгоритмів, наприклад, тих, що застосовуються в комп'ютерній графіці та машинному навчанні.

**Недоліки бінарного пошуку:**

* Потрібно, щоб масив був відсортований. Якщо масив не відсортовано, ми повинні спочатку відсортувати його перед виконанням пошуку. Це додає додаткову часову складність O(n log n) на етапі сортування, що може зробити бінарний пошук менш ефективним для дуже малих масивів.
* Двійковий пошук вимагає, щоб масив, який шукається, зберігався у суміжних комірках пам'яті. Це може бути проблемою, якщо масив занадто великий, щоб поміститися в пам'яті, або якщо масив зберігається на зовнішній пам'яті, наприклад, на жорсткому диску або в хмарі.
* Двійковий пошук вимагає, щоб елементи масиву були порівнянними, тобто щоб їх можна було впорядкувати. Це може бути проблемою, якщо елементи масиву не впорядковані природним чином або якщо порядок не є чітко визначеним.
* Двійковий пошук може бути менш ефективним, ніж інші алгоритми, такі як хеш-таблиці, для пошуку дуже великих наборів даних, які не вміщуються в пам'яті.

**Застосування бінарного пошуку:**

* Пошук у машинному навчанні: Бінарний пошук можна використовувати як будівельний блок для більш складних алгоритмів, що використовуються в машинному навчанні, таких як алгоритми навчання нейронних мереж або пошуку оптимальних гіперпараметрів для моделі.
* Зазвичай використовується в змагальному програмуванні.
* Може використовуватися для пошуку в комп'ютерній графіці. Бінарний пошук може бути використаний як будівельний блок для більш складних алгоритмів, що використовуються в комп'ютерній графіці, таких як алгоритми трасування променів або відображення текстур.
* Може використовуватися для пошуку в базі даних. Бінарний пошук можна використовувати для ефективного пошуку в базі даних записів, наприклад, у базі даних клієнтів або каталозі товарів.

**Висновок:**

* Бінарний пошук - це ефективний алгоритм пошуку елемента у відсортованому масиві.
* Часова складність бінарного пошуку складає O(log n).
* Одним з основних недоліків бінарного пошуку є те, що масив повинен бути відсортований.
* Корисний алгоритм для побудови більш складних алгоритмів у комп'ютерній графіці та машинному навчанні.