**Binary Search**

# Binary search

Binary search, also known as **Half-interval search, Logarithmic search, or Binary chop**, is a search algorithm that finds the position of a target value within a sorted array.

Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with the remaining half being empty, the target is not in the array.

# Basic Algorithm

Given an array A of n elements with values or records A0, A1, ..., An−1, sorted such that A0 ≤ A1 ≤ ... ≤ An−1, and target value T, the following subroutine uses binary search to find the index of T in A.

1. Set L to 0 and R to n − 1.
2. If L > R, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor (the largest previous integer) of (L + R) / 2.
4. If Am < T, set L to m + 1 and go to step 2.
5. If Am > T, set R to m − 1 and go to step 2.
6. Now Am = T, the search is done; return m.

This iterative procedure keeps track of the search boundaries with the two variables. Some implementations may check whether the middle element is equal to the target at the end of the procedure. This results in a faster comparison loop, but requires one more iteration on average.

# Pseudocode

Procedure binary\_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

~~set midPoint = (lowerBound + upperBound) / 2~~ //may cause overflow see [Implementation Issues](#_Implementation_Issues)

if A[midPoint] < x

set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

end procedure

# C/C++ Implementation

C provides the function **bsearch()** in its standard library, which is typically implemented via binary search (although the official standard does not require it so).

C++'s Standard Template Library provides the functions **binary\_search(),** lower\_bound(), upper\_bound() and equal\_range().

// 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;

if (arr[m] == x) // Check if x is present at mid

return m;

if (arr[m] < x) // If x greater, ignore left half

l = m + 1;

else // If x is smaller, ignore right half

r = m - 1;

}

return -1; // if we reach here, then element was not present

}

# Complexity

**Best case**: when the search term would be in the middle of the list.

**Average case**: when the search term is anywhere else in the list other than those mentioned in best and worst case

**Worst case**: when the search term is not in the list, or when the search term is one item away from the middle of the list, or when the search term is the first or last item in the list

\* **Auxiliary Space**:

* O(1) in case of iterative implementation.
* O(logn) recursion call stack space, In case of recursive implementation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average case** | **Worst case** | **Space Complexity** |
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(logn) | O(logn) | O(1)\* |
| Jump Search | O(1) | O(√ n) | O(√ n) | O(1) |
| Interpolation Search | O(1) | O (log log n)) | O(n) | O(1) |
| Exponential Search | O(1) | O(log i) | O(log i) | O(1) |
| Fibonacci Search | O(1) | O(logn) | O(logn) | O(1) |

The time complexity of Binary Search can be written as

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

# Variations

1. Uniform binary search
2. Exponential search
3. Interpolation search
4. Fractional cascading
5. Fibonacci search
6. Noisy binary search
7. Quantum binary search

# Implementation Issues

## Overflow Error

Jon Bentley's own implementation of binary search, published in his 1986 book Programming Pearls, contained an overflow error that remained undetected for over twenty years. The Java programming language library implementation of binary search had the same overflow bug for more than nine years.

In a practical implementation, the variables used to represent the indices will often be of fixed size, and this can result in an arithmetic overflow for very large arrays. If the midpoint of the span is calculated as (L + R) / 2, then the value of L + R may exceed the range of integers of the data type used to store the midpoint, even if L and R are within the range. If L and R are nonnegative, this can be avoided by calculating the midpoint as L + (R − L) / 2.

**midpoint = L + (R − L) / 2 // correct way to calculate mid point**

midpoint = (L + R) >>> 1; **// in Java**

midpoint = ((unsigned int)L + (unsigned int)R)) >> 1; **// in C++**

~~midpoint = (L + R) / 2~~ // the value of L + R may exceed the range of integers of the data type causing arithmetic overflow

## Infinite Loop Error

1. If the target value is greater than the greatest value in the array, and the last index of the array is the maximum representable value of L, the value of L will eventually become too large and overflow.
2. A similar problem will occur if the target value is smaller than the least value in the array and the first index of the array is the smallest representable value of R. In particular, this means that R must not be an unsigned type if the array starts with index 0.

An infinite loop may occur if the exit conditions for the loop are not defined correctly. Once L exceeds R, the search has failed and must convey the failure of the search.

In addition, the loop must be exited when the target element is found, or in the case of an implementation where this check is moved to the end, checks for whether the search was successful or failed at the end must be in place. Bentley found that, in his assignment of binary search, most of the programmers who implemented binary search incorrectly made an error defining the exit conditions.

# Application

1. to figure out the bad commit in git
2. debugging a somewhat linear piece of code. if the code has many steps mostly executed in a sequence and there's a bug, you can isolate the bug by finding the earliest step where the code produces results which are different from the expected ones.
3. cherry picking a bad code change from a release candidate. When pushing a large code release in production one would sometimes find that there's a problem with that binary. If reverting the whole release wasn't an option the release engineer would binary search through the code change ids. He would figure out the earliest code change which creates the bug.
4. figuring out resource requirements for a large system. one could try running load tests on the system and binary search for the minimum amount of CPUs required to handle a predicted load. (this approach is better than random guessing but much worse than doing some analysis of your system and doing some good educated guesses)

# Example

## Iterative Binary Search

#include <stdio.h>

// 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;

if (arr[m] == x) // Check if x is present at mid

return m;

if (arr[m] < x) // If x greater, ignore left half

l = m + 1;

else // If x is smaller, ignore right half

r = m - 1;

}

return -1; // if we reach here, then element was not present

}

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;

}

Output:

Element is present at index 3

## 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 (arr[mid] == x)

return mid;

if (arr[mid] > x)

return binarySearch(arr, l, mid-1, x);

else

return binarySearch(arr, mid+1, r, x);

}

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;

}

Output:

Element is present at index 3