= Binary search algorithm =

In computer science , binary search , also known as half @-@ interval search or logarithmic search , is a search algorithm that finds the position of a target value within a sorted array . It 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 .

Binary search runs in at worst logarithmic time , making

O
(
log
?
n
)

{ \ textstyle O (\ log n) }
comparisons , where

n

{ \ textstyle n }
is the number of elements in the array and

log

{ \ textstyle \ log }
is the binary logarithm ; and using only constant

(
O
(
1
)
)

{ \ textstyle (O (1)) }
space . Although specialized data structures designed for fast searching ? such as hash tables ? can be searched more efficiently , binary search applies to a wider range of search problems .

Although the idea is simple , implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation .

There exist numerous variations of binary search . One variation in particular (fractional cascading

) speeds up binary searches for the same value in multiple arrays .

= = Algorithm = =

Binary search works on sorted arrays . A binary search begins by comparing the middle element of the array with the target value . If the target value matches the middle element , its position in the array is returned . If the target value is less or more than the middle element , the search continues the lower or upper half of the array respectively with a new middle element , eliminating the other half from consideration .

= = = Procedure = = =

Given an array A of n elements with values or records A0 ... An ? 1 and target value T , the following subroutine uses binary search to find the index of T in A.
Set L to 0 and R to n ? 1 .
If L > R , the search terminates as unsuccessful . Set m (the position of the middle element) to the floor of (L + R) / 2 .
If Am < T , set L to m + 1 and go to step 2 .
If Am > T , set R to m ? 1 and go to step 2 .
If Am = T , the search is done ; return m .
This iterative procedure keeps track of the search boundaries via two variables ; a recursive version would keep its boundaries in its recursive calls . Some implementations may place the comparison for equality at the end of the algorithm , resulting in a faster comparison loop but costing one more iteration on average .

= = = Approximate matches = = =

The above procedure only performs exact matches , finding the position of a target value . However , due to the ordered nature of sorted arrays , it is trivial to extend binary search to perform approximate matches . Particularly , binary search can be used to compute , relative to a value , its rank (the number of smaller elements) , predecessor (next @-@ smallest element) , successor (next @-@ largest element) , and nearest neighbor . Range queries seeking the number of elements between two values can be performed with two rank queries .
Rank queries can be performed using a modified version of binary search . By returning m on a successful search , and L on an unsuccessful search , the number of elements less than the target value is returned instead .
Predecessor and successor queries can be performed with rank queries . Once the rank of the target value is known , its predecessor is the element at the position given by its rank (as it is the largest element that is smaller than the target value) . Its successor is the element after it (if it is present in the array) or at the next position after the predecessor (otherwise) . The nearest neighbor of the target value is either its predecessor or successor , whichever is closer .
Range queries are also straightforward . Once the ranks of the two values are known , the number of elements greater than or equal to the first value and less than the second is the difference of the two ranks . This count can be adjusted up or down by one according to whether the endpoints of the range should be considered to be part of the range and whether the array contains keys matching those endpoints .

= = Performance = =

The performance of binary search can be analyzed by reducing the procedure to a binary comparison tree , where the root node is the middle element of the array ; the middle element of the lower half is left of the root and the middle element of the upper half is right of the root . The rest of the tree is built in a similar fashion . This model represents binary search ; starting from the root

node , the left or right subtrees are traversed depending on whether the target value is less or more than the node under consideration , representing the successive elimination of elements .
The worst case is

?
log
?
n
+
1
?

{ \ textstyle \ lfloor \ log n + 1 \ rfloor }
iterations (of the comparison loop) , where the

?
?

{ \ textstyle \ lfloor \ rfloor }
notation denotes the floor function that rounds its argument down to an integer . This is reached when the search reaches the deepest level of the tree , equivalent to a binary search that has reduced to one element and , in each iteration , always eliminates the smaller subarray out of the two if they are not of equal size .
On average , assuming that each element is equally likely to be searched , by the time the search completes , the target value will most likely be found at the second @-@ deepest level of the tree . This is equivalent to a binary search that completes one iteration before the worst case , reached after

log
?

n

?
1

{ \ textstyle \ log { n } -1 }
iterations . However , the tree may be unbalanced , with the deepest level partially filled , and equivalently , the array may not be divided perfectly by the search in some iterations , half of the time resulting in the smaller subarray being eliminated . The actual number of average iterations is slightly higher , at

log
?
n
?

n
?
log
?
n
?
1

n

{ \ textstyle \ log n- { \ frac { n- \ log n @-@ 1 } { n } } }
iterations . In the best case , where the first middle element selected is equal to the target value , its position is returned after one iteration . In terms of iterations , no search algorithm that is based solely on comparisons can exhibit better average and worst @-@ case performance than binary search .

Each iteration of the binary search algorithm defined above makes one or two comparisons , checking if the middle element is equal to the target value in each iteration . Again assuming that each element is equally likely to be searched , each iteration makes 1 @.@ 5 comparisons on average . A variation of the algorithm instead checks for equality at the very end of the search , eliminating on average half a comparison from each iteration . This decreases the time taken per iteration very slightly on most computers , while guaranteeing that the search takes the maximum number of iterations , on average adding one iteration to the search . Because the comparison loop is performed only

?
log
?
n
+
1
?

{ \ textstyle \ lfloor \ log n + 1 \ rfloor }
times in the worst case , for all but enormous

n

{ \ textstyle n }
, the slight increase in comparison loop efficiency does not compensate for the extra iteration . Knuth 1998 gives a value of

2

66

{ \ textstyle 2 ^ { 66 } }
(more than 73 quintillion) elements for this variation to be faster .
The iterative version of binary search only requires three extra variables , taking constant extra space . When compiled with a compiler that does not support tail call elimination , the recursive version takes space proportional to the number of iterations as extra stack space is required to store the variables . If the array is passed by value instead of by reference to the binary search subroutine , the original array will have to be copied , costing

O
(
n
)

{ \ textstyle O (n) }
extra time and space . Since binary search does not modify the elements of the array , it is safe to pass the array by reference .
Fractional cascading can be used to speed up searches of the same value in multiple arrays . Where

k

{ \ textstyle k }
is the number of arrays , searching each array for the target value takes

O
(
k
log
?
n
)

{ \ textstyle O (k \ log n) }
time ; fractional cascading reduces this to

O
(
k
+
log
?
n
)

{ \ textstyle O (k + \ log n) }
.

= = Binary search versus other schemes = =

= = = Hash tables = = =

For implementing associative arrays , hash tables , a data structure that maps keys to records using a hash function , are generally faster than binary search on a sorted array of records ; most implementations require only amortized constant time on average . However , hashing is not useful for approximate matches , such as computing the next @-@ smallest , next @-@ largest , and nearest key , as the only information given on a failed search is that the target is not present in any record . Binary search is ideal for such matches , performing them in logarithmic time . In addition , all operations possible on a sorted array can be performed ? such as finding the smallest and largest key and performing range searches .

= = = Binary search trees = = =

A binary search tree is a binary tree data structure that works based on the principle of binary search : the records of the tree are arranged in sorted order , and traversal of the tree is performed using a logarithmic time binary search @-@ like algorithm . Insertion and deletion also require logarithmic time in binary search trees . This is faster than the linear time insertion and deletion of sorted arrays , and binary trees retain the ability to perform all the operations possible on a sorted array .

However , binary search is usually more efficient for searching as binary search trees will most likely be imperfectly balanced , resulting in slightly worse performance than binary search . This applies even to balanced binary search trees , binary search trees that balance their own nodes ? as they rarely produce optimally @-@ balanced trees ? but to a lesser extent . Although unlikely , the tree may be severely imbalanced with few internal nodes with two children , resulting in the average and worst @-@ case search time approaching

n

{ \ textstyle n }
comparisons . Binary search trees take more space than sorted arrays .

= = = Linear search = = =

Linear search is a simple search algorithm that checks every record until it finds the target value . Linear search can be done on a linked list , which allows for faster insertion and deletion than an array . Binary search is faster than linear search for sorted arrays except if the array is short . If the array must first be sorted , that cost must be amortized (spread) over any searches . Sorting the array also enables efficient approximate matches and other operations .

= = = Other data structures = = =

There exist data structures that may beat binary search in some cases for both searching and other operations available for sorted arrays . For example , searches , approximate matches , and the operations available to sorted arrays can be performed more efficiently than binary search on specialized data structures such as van Emde Boas trees , fusion trees , tries , and bit arrays . However , while these operations can always be done at least efficiently on a sorted array regardless of the keys , such data structures are usually only faster because they exploit the properties of keys with a certain attribute (usually keys that are small integers) , and thus will be time or space consuming for keys that do not have that attribute .

= = Variations = =

= = = Uniform binary search = = =

Uniform binary search stores , instead of the lower and upper bounds , the index of the middle element and the number of elements around the middle element that were not eliminated yet . Each step reduces the width by about half . This variation is uniform because the difference between the indices of middle elements and the preceding middle elements chosen remains constant between searches of arrays of the same length .

= = = Fibonacci search = = =

Fibonacci search is a method similar to binary search that successively shortens the interval in which the maximum of a unimodal function lies . Given a finite interval , a unimodal function , and the maximum length of the resulting interval , Fibonacci search finds a Fibonacci number such that if the interval is divided equally into that many subintervals , the subintervals would be shorter than the maximum length . After dividing the interval , it eliminates the subintervals in which the maximum cannot lie until one or more contiguous subintervals remain .

= = = Exponential search = = =

Exponential search is an algorithm for searching primarily in infinite lists , but it can be applied to select the upper bound for binary search . It starts by finding the first element with an index that is both a power of two and greater than the target value . Afterwards , it sets that index as the upper bound , and switches to binary search . A search takes

?

log
?

x

+
1
?

{ \ textstyle \ lfloor \ log { x } + 1 \ rfloor }
iterations of the exponential search and at most

?
log
?

n

?

{ \ textstyle \ lfloor \ log { n } \ rfloor }
iterations of the binary search , where

x

{ \ textstyle x }
is the position of the target value . Only if the target value is near the beginning of the array is this variation more efficient than selecting the highest element as the upper bound .

= = = Interpolation search = = =

Instead of merely calculating the midpoint , interpolation search attempts to calculate the position of the target value , taking into account the lowest and highest elements in the array and the length of the array . This is only possible if the array elements are numbers . It works on the basis that the midpoint is not the best guess in many cases ; for example , if the target value is close to the highest element in the array , it is likely to be located near the end of the array . When the distribution of the array elements is uniform or near uniform , it makes

O
(
log
?
log
?

n
)

{ \ textstyle O (\ log \ log n) }
comparisons .
In practice , interpolation search is slower than binary search for small arrays , as interpolation search requires extra computation , and the slower growth rate of its time complexity compensates for this only for large arrays .

= = = Fractional cascading = = =

Fractional cascading is a technique that speeds up binary searches for the same element for both exact and approximate matching in " catalogs " (arrays of sorted elements) associated with vertices in graphs . Searching each catalog separately requires

O
(
k
log
?
n
)

{ \ textstyle O (k \ log n) }
time , where

k

{ \ textstyle k }
is the number of catalogs . Fractional cascading reduces this to

O
(
k
+
log
?
n
)

{ \ textstyle O (k + \ log n) }
by storing specific information in each catalog about other catalogs .
Fractional cascading was originally developed to efficiently solve various computational geometry

problems , but it also has been applied elsewhere , in domains such as data mining and Internet Protocol routing .

= = History = =

In 1946 , John Mauchly made the first mention of binary search as part of the Moore School Lectures , the first ever set of lectures regarding any computer @-@ related topic . Every published binary search algorithm worked only for arrays whose length is one less than a power of two until 1960 , when Derrick Henry Lehmer published a binary search algorithm that worked on all arrays . In 1962 , Hermann Bottenbruch presented an ALGOL 60 implementation of binary search that placed the comparison for equality at the end , increasing the average number of iterations by one , but reducing to one the number of comparisons per iteration . The uniform binary search was presented to Donald Knuth in 1971 by A. K. Chandra of Stanford University and published in Knuth 's The Art of Computer Programming . In 1986 , Bernard Chazelle and Leonidas J. Guibas introduced fractional cascading , a technique used to speed up binary searches in multiple arrays .

= = Implementation issues = =

Although the basic idea of binary search is comparatively straightforward , the details can be surprisingly tricky ... ? Donald Knuth

When Jon Bentley assigned binary search as a problem in a course for professional programmers , he found that ninety percent failed to provide a correct solution after several hours of working on it , and another study published in 1988 shows that accurate code for it is only found in five out of twenty textbooks . Furthermore , 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 . This can be avoided by calculating the midpoint as L + (R ? L) / 2 .

If the target value is greater than the highest value in the array , and the size of the array is the maximum representable integer , then an overflow will result if one @-@ based instead of zero @-@ based indexing is used . With one @-@ based indexing , R is set to the length to the array instead of one less than the length . Eventually L will equal R after the lower elements are eliminated , while R does not change . Because the target value exceeds the highest value in the array , the next iteration will set L to R + 1 . This results in an overflow because L exceeds R , which is equal to the maximum representable integer . This can be avoided by using zero @-@ based indexing or a larger data type for L so it can fit R + 1 .

An infinite loop may occur if the exit conditions for the loop ? or equivalently , recursive step ? 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 , this error was made by most of the programmers who failed to implement a binary search correctly .

= = Library support = =

Many languages ' standard libraries include binary search routines :

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 STL provides the functions binary _ search ( ) , lower _ bound ( ) , upper _ bound ( ) and equal _ range ( ) .
Java offers a set of overloaded binarySearch ( ) static methods in the classes Arrays and Collections in the standard java.util package for performing binary searches on Java arrays and on Lists , respectively .
Microsoft 's .NET Framework 2 @.@ 0 offers static generic versions of the binary search algorithm in its collection base classes . An example would be System.Array 's method BinarySearch < T > (T [ ] array , T value) .
Python provides the bisect module .
Ruby 's Array class includes a bsearch method with built @-@ in approximate matching .
Go 's sort standard library package contains the functions Search , SearchInts , SearchFloat64s , and SearchStrings , which implement general binary search , as well as specific implementations for searching slices of integers , floating @-@ point numbers , and strings , respectively .
For Objective @-@ C , the Cocoa framework provides the NSArray -indexOfObject : inSortedRange : options : usingComparator : method in Mac OS X 10 @.@ 6 + . Apple 's Core Foundation C framework also contains a CFArrayBSearchValues ( ) function .

= = = Works = = =