



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Azərbaycanca
Български
Bosanski
Čeština
Deutsch
Ελληνικά
فارسی
Français
한국어
हिन्दी
Bahasa Indonesia
Interlingua
Italiano
עברית
Македонски
नेपाली
日本語
Norsk bokmål
Polski
Português
Română
Русский
Slovenčina
Slovenščina
Српски / srpski
Suomi
Svenska
தமிழ்
Türkçe

Create account Log in

Article Talk

Read Edit View history

Search

Binary search algorithm

From Wikipedia, the free encyclopedia

This article is about searching a finite sorted array. For searching continuous function values, see [bisection method](#).



This article **may require [cleanup](#) to meet Wikipedia's [quality standards](#)**. No [cleanup reason](#) has been specified. Please help [improve this article](#) if you can. *(April 2011)*

In [computer science](#), a **binary search** or **half-interval search algorithm** finds the position of a target value within a [sorted array](#).^{[1][2]} The binary search algorithm can be classified as a [dichotomic divide-and-conquer search algorithm](#) and executes in logarithmic time.

Contents [hide]

- Overview
- Example
 - Number guessing game
 - Word lists
 - Applications to complexity theory
- Algorithm
 - Recursive
 - Iterative
 - Deferred detection of equality
- Performance
 - Average performance
- Variations
 - Exclusive or inclusive bounds
 - Midpoint and width
 - Search domain
 - Noisy search
 - Exponential search
 - Interpolated search
- Implementation issues
 - Arithmetic
- Language support
- See also
- References
 - Other sources
- External links

Binary search algorithm

Class	Search algorithm
Data structure	Array
Worst case performance	$O(\log n)$
Best case performance	$O(1)$
Average case performance	$O(\log n)$
Worst case space complexity	$O(1)$

Overview [edit]

The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished. If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

Example [edit]

```
Sorted array: L = [1, 3, 4, 6, 8, 9, 11]
Target value: X = 4
Compare X to 6. X is smaller. Repeat with L = [1, 3, 4].
Compare X to 3. X is larger. Repeat with L = [4].
Compare X to 4. X equals 4, so the position is returned.
```

Number guessing game [\[edit\]](#)

This rather simple game begins something like "I'm thinking of an integer between forty and sixty inclusive, and to your guesses I'll respond 'Higher', 'Lower', or 'Yes!' as might be the case." Supposing that N is the number of possible values (here, twenty-one, as "inclusive" was stated), then at most $\lceil \log_2 N \rceil$ questions are required to determine the number, since each question halves the search space. Note that one less question (iteration) is required than for the general algorithm, since the number is already constrained to be within a particular range.

Even if the number to guess can be arbitrarily large, in which case there is no upper bound N , the number can be found in at most $2\lceil \log_2 k \rceil + 1$ steps (where k is the (unknown) selected number) by first finding an upper bound with [one-sided binary search](#).^{[\[citation needed\]](#)} For example, if the number were 11, the following sequence of guesses could be used to find it: 1 (Higher), 2 (Higher), 4 (Higher), 8 (Higher), 16 (Lower), 12 (Lower), 10 (Higher). Now we know that the number must be 11 because it is higher than 10 and lower than 12.

One could also extend the method to include negative numbers; for example the following guesses could be used to find -13 : 0, -1 , -2 , -4 , -8 , -16 , -12 , -14 . Now we know that the number must be -13 because it is lower than -12 and higher than -14 .

Word lists [\[edit\]](#)

People typically use a mixture of the binary search and [interpolative search](#) algorithms when searching a [telephone book](#), after the initial guess we exploit the fact that the entries are sorted and can rapidly find the required entry. For example when searching for Smith, if Rogers and Thomas have been found, one can flip to a page about halfway between the previous guesses. If this shows Samson, it can be concluded that Smith is somewhere between the Samson and Thomas pages so these can be divided.

Applications to complexity theory [\[edit\]](#)

Even if we do not know a fixed range the number k falls in, we can still determine its value by asking $2\lceil \log_2 k \rceil$ simple yes/no questions of the form "Is k greater than x ?" for some number x . As a simple consequence of this, if you can answer the question "Is this integer property k greater than a given value?" in some amount of time then you can find the value of that property in the same amount of time with an added factor of $\log_2 k$. This is called a [reduction](#), and it is because of this kind of reduction that most complexity theorists concentrate on [decision problems](#), algorithms that produce a simple yes/no answer.

For example, suppose we could answer "Does this $n \times n$ matrix have [permanent](#) larger than k ?" in $O(n^2)$ time. Then, by using binary search, we could find the (ceiling of the) permanent itself in $O(n^2 \log p)$ time, where p is the value of the permanent. Notice that p is not the size of the input, but the *value* of the output; given a matrix whose maximum item (in absolute value) is m , p is bounded by $m^n n!$. Hence $\log p = O(n \log n + n \log m)$. A binary search could find the permanent in $O(n^3 \log n + n^3 \log m)$.

Algorithm [\[edit\]](#)

Recursive [\[edit\]](#)

A straightforward implementation of binary search is [recursive](#). The initial call uses the indices of the entire array to be searched. The procedure then calculates an index midway between the two indices, determines which of the two subarrays to search, and then does a recursive call to search that subarray. Each of the calls is [tail recursive](#), so a compiler need not make a new stack frame for each call. The variables `imin` and `imax` are the lowest and highest inclusive indices that are searched.

```
int binary_search(int A[], int key, int imin, int imax)
{
    // test if array is empty
    if (imax < imin)
        // set is empty, so return value showing not found
        return KEY_NOT_FOUND;
    else
```

```

{
    // calculate midpoint to cut set in half
    int imid = midpoint(imin, imax);

    // three-way comparison
    if (A[imid] > key)
        // key is in lower subset
        return binary_search(A, key, imin, imid - 1);
    else if (A[imid] < key)
        // key is in upper subset
        return binary_search(A, key, imid + 1, imax);
    else
        // key has been found
        return imid;
}
}

```

It is invoked with initial `imin` and `imax` values of `0` and `N-1` for a zero based array of length `N`.

The number type "int" shown in the code has an influence on how the midpoint calculation can be implemented correctly. With unlimited numbers, the midpoint can be calculated as `"(imin + imax) / 2"`. In practical programming, however, the calculation is often performed with numbers of a limited range, and then the intermediate result `"(imin + imax)"` might overflow. With limited numbers, the midpoint can be calculated correctly as `"imin + ((imax - imin) / 2)"`.

Iterative [\[edit\]](#)

The binary search algorithm can also be expressed iteratively with two index limits that progressively narrow the search range.^[3]

```

int binary_search(int A[], int key, int imin, int imax)
{
    // continue searching while [imin,imax] is not empty
    while (imin <= imax)
    {
        // calculate the midpoint for roughly equal partition
        int imid = midpoint(imin, imax);
        if (A[imid] == key)
            // key found at index imid
            return imid;
        // determine which subarray to search
        else if (A[imid] < key)
            // change min index to search upper subarray
            imin = imid + 1;
        else
            // change max index to search lower subarray
            imax = imid - 1;
    }
    // key was not found
    return KEY_NOT_FOUND;
}

```

Deferred detection of equality [\[edit\]](#)

The above iterative and recursive versions take three paths based on the key comparison: one path for less than, one path for greater than, and one path for equality. (There are two conditional branches.) The path for equality is taken only when the record is finally matched, so it is rarely taken. That branch path can be moved outside the search loop in the deferred test for equality version of the algorithm. The following algorithm uses only one conditional branch per iteration.^[4]

```

// inclusive indices
// 0 <= imin when using truncate toward zero divide
// imid = (imin+imax)/2;
// imin unrestricted when using truncate toward minus infinity divide
// imid = (imin+imax)>>1; or
// imid = (int)floor((imin+imax)/2.0);
int binary_search(int A[], int key, int imin, int imax)

```

```

{
    // continually narrow search until just one element remains
    while (imin < imax)
    {
        int imid = midpoint(imin, imax);

        // code must guarantee the interval is reduced at each iteration
        assert(imid < imax);
        // note: 0 <= imin < imax implies imid will always be less than imax

        // reduce the search
        if (A[imid] < key)
            imin = imid + 1;
        else
            imax = imid;
    }
    // At exit of while:
    //   if A[] is empty, then imax < imin
    //   otherwise imax == imin

    // deferred test for equality
    if ((imax == imin) && (A[imin] == key))
        return imin;
    else
        return KEY_NOT_FOUND;
}

```

The deferred detection approach foregoes the possibility of early termination on discovery of a match, so the search will take about $\log_2(N)$ iterations. On average, a *successful* early termination search will not save many iterations. For large arrays that are a power of 2, the savings is about two iterations. Half the time, a match is found with one iteration left to go; one quarter the time with two iterations left, one eighth with three iterations, and so forth. The infinite series sum is 2.

The deferred detection algorithm has the advantage that if the keys are not unique, it returns the smallest index (the starting index) of the region where elements have the search key. The early termination version would return the first match it found, and that match might be anywhere in region of equal keys.

Performance [\[edit\]](#)

With each test that fails to find a match at the probed position, the search is continued with one or other of the two sub-intervals, each at most half the size. More precisely, if the number of items, N , is odd then both sub-intervals will contain $(N-1)/2$ elements, while if N is even then the two sub-intervals contain $N/2-1$ and $N/2$ elements.

If the original number of items is N then after the first iteration there will be at most $N/2$ items remaining, then at most $N/4$ items, at most $N/8$ items, and so on. In the worst case, when the value is not in the list, the algorithm must continue iterating until the span has been made empty; this will have taken at most $\lfloor \log_2(N) \rfloor + 1$ iterations, where the $\lfloor \cdot \rfloor$ notation denotes the [floor function](#) that rounds its argument down to an integer. This [worst case analysis](#) is tight: for any N there exists a query that takes exactly $\lfloor \log_2(N) \rfloor + 1$ iterations. When compared to [linear search](#), whose worst-case behaviour is N iterations, we see that binary search is substantially faster as N grows large. For example, to search a list of one million items takes as many as one million iterations with linear search, but never more than twenty iterations with binary search. However, a binary search can only be performed if the list is in sorted order.

Average performance [\[edit\]](#)

$\log_2(N)-1$ is the expected number of probes in an average successful search, and the worst case is $\log_2(N)$, just one more probe.^{[\[citation needed\]](#)} If the list is empty, no probes at all are made. Thus binary search is a [logarithmic algorithm](#) and executes in $O(\log N)$ time. In most cases it is considerably faster than a [linear search](#). It can be implemented using [iteration](#), or [recursion](#). In some languages it is more elegantly expressed recursively; however, in some C-based languages tail recursion is not eliminated and the recursive version requires more stack space.

Binary search can interact poorly with the memory hierarchy (i.e. [caching](#)), because of its random-access nature. For in-memory searching, if the span to be searched is small, a linear search may have superior performance simply because it exhibits better [locality of reference](#). For external searching, care must be taken or each of the first several probes will lead to a disk seek. A common method is to abandon binary searching for

linear searching as soon as the size of the remaining span falls below a small value such as 8 or 16 or even more in recent computers. The exact value depends entirely on the machine running the algorithm.

Notice that for multiple searches *with a fixed value for N* , then (with the appropriate regard for integer division), the first iteration always selects the middle element at $N/2$, and the second always selects either $N/4$ or $3N/4$, and so on. Thus if the array's key values are in some sort of slow storage (on a disc file, in virtual memory, not in the CPU's on-chip memory), keeping those three keys in a local array for a special preliminary search will avoid accessing widely separated memory. Escalating to seven or fifteen such values will allow further levels at not much cost in storage. On the other hand, if the searches are frequent and not separated by much other activity, the computer's various storage control features will more or less automatically promote frequently accessed elements into faster storage.

When multiple binary searches are to be performed for the same key in related lists, [fractional cascading](#) can be used to speed up successive searches after the first one.

In theory binary search is usually faster than linear search, but in practice that may not hold true. For small arrays (say about 64 items or less), linear search may have better performance. For any size unsorted array, the cost of sorting the array may exceed the speed advantage of binary search when the array is only searched a few times because the time to sort the array is comparable to $\log(n)$ linear searches^[5] For example, if an unsorted array will only be searched once, it will be faster to just do a linear search rather than sorting the array and then doing a binary search.

Variations [\[edit\]](#)

Exclusive or inclusive bounds [\[edit\]](#)

The most significant differences are between the "exclusive" and "inclusive" forms of the bounds. In the "exclusive" bound form the span to be searched is $(L+1)$ to $(R-1)$, and this may seem clumsy when the span to be searched could be described in the "inclusive" form, as L to R . Although the details differ the two forms are equivalent as can be seen by transforming one version into the other. The inclusive bound form can be attained by replacing all appearances of " L " by " $(L-1)$ " and " R " by " $(R+1)$ " then rearranging. Thus, the initialisation of $L := 0$ becomes $(L-1) := 0$ or $L := 1$, and $R := N+1$ becomes $(R+1) := N+1$ or $R := N$. So far so good, but note now that the changes to L and R are no longer simply transferring the value of p to L or R as appropriate but now must be $(R+1) := p$ or $R := p-1$, and $(L-1) := p$ or $L := p+1$.

Thus, the gain of a simpler initialisation, done once, is lost by a more complex calculation, and which is done for every iteration. If that is not enough, the test for an empty span is more complex also, as compared to the simplicity of checking that the value of p is zero. Nevertheless, the inclusive bound form is found in many publications, such as [Donald Knuth](#). *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition.

Another common variation uses inclusive bounds for the left bound, but exclusive bounds for the right bound. This is derived from the fact that the bounds in a language with zero-based arrays can be simply initialized to 0 and the size of the array, respectively. This mirrors the way array slices are represented in some programming languages.

Midpoint and width [\[edit\]](#)

A different variation involves abandoning the L and R pointers and using a current position p and a width w . At each iteration, the position p is adjusted and the width w is halved. Knuth states, "It is possible to do this, but only if extreme care is paid to the details."^[6]

Search domain [\[edit\]](#)

There is no particular requirement that the array being searched has the bounds 1 to N . It is possible to search a specified range, elements *first* to *last* instead of 1 to N . All that is necessary is that the initialization of the bounds be $L := \text{first}-1$ and $R := \text{last}+1$, then all proceeds as before.

The elements of the list are not necessarily all unique. If one searches for a value that occurs multiple times in the list, the index returned will be of the first-encountered equal element, and this will not necessarily be that of the first, last, or middle element of the run of equal-key elements but will depend on the positions of the values. Modifying the list even in seemingly unrelated ways such as adding elements elsewhere in the list may change the result.

If the location of the first and/or last equal element needs to be determined, this can be done efficiently with a variant of the binary search algorithms which perform only one inequality test per iteration. See [deferred](#)

[detection of equality](#).

Noisy search [\[edit\]](#)

Several algorithms closely related to or extending binary search exist. For instance, **noisy binary search** solves the same class of problems as regular binary search, with the added complexity that any given test can return a false value at random. (Usually, the number of such erroneous results are bounded in some way, either in the form of an average error rate, or in the total number of errors allowed per element in the search space.) Optimal algorithms for several classes of noisy binary search problems have been known since the late seventies, and more recently, optimal algorithms for noisy binary search in quantum computers (where several elements can be tested at the same time) have been discovered.

Exponential search [\[edit\]](#)

An [exponential search](#) (also called a **one-sided search**) searches from a starting point within the array and either expects that the element *p* being sought is nearby or the upper (lower) bound on the array is unknown. Starting with a step size of 1 and doubling with each step, the method looks for a number \geq (\leq) *p*. Once the upper (lower) bound is found, then the method proceeds with a binary search. The complexity of the search is $2\lceil\log_2 n\rceil$ if the sought element is in the *n*th array position. This depends only on *n* and not on the size of the array.

Interpolated search [\[edit\]](#)

Main article: [Interpolation search](#)

An interpolated search tries to guess the location of the element *p* you're searching for, typically by calculating a midpoint based on the lowest and highest value and assuming a fairly even distribution of values. When *p* has been determined an exponential search is performed.^{[\[clarification needed\]](#)}

Implementation issues [\[edit\]](#)

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky... — [Donald Knuth](#)^{[\[7\]](#)}

When [Jon Bentley](#) assigned it as a problem in a course for professional programmers, he found that an astounding ninety percent failed to code a binary search correctly after several hours of working on it,^{[\[8\]](#)} and another study shows that accurate code for it is only found in five out of twenty textbooks.^{[\[9\]](#)} Furthermore, Bentley's own implementation of binary search, published in his 1986 book *Programming Pearls*, contains an error that remained undetected for over twenty years.^{[\[10\]](#)}

Arithmetic [\[edit\]](#)

In a practical implementation, the variables used to represent the indices will often be of finite size, hence only capable of representing a finite range of values. For example, 32-bit [unsigned integers](#) can only hold values from 0 to 4294967295. 32-bit signed integers can only hold values from -2147483648 to 2147483647. If the binary search algorithm is to operate on large arrays, this has three implications:

- The values `first - 1` and `last + 1` must both be representable within the finite bounds of the chosen integer type. Therefore, continuing the 32-bit unsigned example, the largest value that `last` may take is +4294967294, not +4294967295. A problem exists even for the "inclusive" form of the method, as if `x > A(4294967295).Key`, then on the final iteration the algorithm will attempt to store 4294967296 into `L` and fail. Equivalent issues apply to the lower limit, where `first - 1` could become negative as when the first element of the array is at index zero.
- If the midpoint of the span is calculated as `p := (L + R) / 2`, then the value `(L + R)` will exceed the number range if `last` is greater than (for unsigned) 4294967295/2 or (for signed) 2147483647/2 and the search wanders toward the upper end of the search space. This can be avoided by performing the calculation as `p := (R - L) / 2 + L`. For example, this bug existed in Java SDK at `Arrays.binarySearch()` from 1.2 to 5.0 and was fixed in 6.0.^{[\[11\]](#)}
- KEY_NOT_FOUND must be a valid value of the return type, but this value can never be an index of the array

Language support [\[edit\]](#)

Many standard libraries provide a way to do a binary search:

- C provides the algorithm function `bsearch()` in its [standard library](#).
- C++'s STL provides the [algorithm functions](#) `binary_search()`, `lower_bound()` and `upper_bound()`.
- 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 `List`s, respectively. They must be arrays of primitives, or the arrays or Lists must be of a type that implements the `Comparable` interface, or you must specify a custom `Comparator` object.
- 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.
- COBOL can perform binary search on internal tables using the `SEARCH ALL` statement.
- Perl can perform a generic binary search using the [CPAN](#) module `List::BinarySearch`.^[12]
- Ruby's `Array` class has included a [bsearch](#) method since version 2.0.0.
- 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.^[13]
- 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.

See also [\[edit\]](#)


- [Interpolation search](#), similar method with better average complexity
- [Index \(information technology\)](#), very fast 'lookup' using an index to directly select an entry
- [Branch table](#), alternative indexed 'lookup' method for decision making
- [Self-balancing binary search tree](#)
- [Run-time analysis](#), illustrating binary search method on machines of differing speeds
- [Bisection method](#), the same idea used to solve equations in the real numbers







References [\[edit\]](#)

- ↑ Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). *Introduction to Algorithms* (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8.
- ↑ Weisstein, Eric W., "Binary Search" [↗](#), *MathWorld*.
- ↑ Press, William H.; Flannery, Brian P.; Teukolsky, Saul A.; Vetterling, William T. (1988), *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, pp. 98–99, ISBN 0-521-35465-X
- ↑ Wirth, Niklaus (1983), *Programming in Modula-2*, p. 35
- ↑ Hovath, Adam (February 5, 2012). "Binary search and linear search performance on the .NET and Mono platform" [↗](#).
- ↑ Knuth, Donald (1968). *Sorting and Searching*. *The Art of Computer Programming* **3** (3rd ed.). Addison-Wesley. p. 414. ISBN 978-0321751041.
- ↑ Knuth, Donald (1997). "Section 6.2.1: Searching an Ordered Table". *Sorting and Searching*. *The Art of Computer Programming* **3** (3rd ed.). Addison-Wesley. pp. 409–426. ISBN 0-201-89685-0.
- ↑ Bentley, Jon (2000) [1986]. *Programming Pearls* (2nd ed.). Addison-Wesley. p. 341. ISBN 0-201-65788-0.
- ↑ Pattis, Richard E. (1988). "Textbook errors in binary searching". *SIGCSE Bulletin* **20**: 190–194. doi:10.1145/52965.53012 [↗](#). cited at Kruse, Robert (1998). *Data Structures and Program Design in C++*. Prentice Hall. p. 280. ISBN 0-13-768995-0.
- ↑ Bloch, Joshua (June 2, 2006) [Updated 17 Feb 2008]. "Extra, Extra – Read All About It: Nearly All Binary Searches and Mergesorts are Broken" [↗](#). *Google Research Blog*.
- ↑ "Bug ID: 5045582 (coll) binarySearch() fails for size larger than 1<<30" [↗](#). *Java Bug Database*. Oracle. 11 May 2004.
- ↑ "List::BinarySearch" [↗](#). CPAN.
- ↑ "Package sort" [↗](#). *The Go Programming Language*.

Other sources [edit]

- Kruse, Robert L.: "Data Structures and Program Design in C++", Prentice-Hall, 1999, [ISBN 0-13-768995-0](#), page 280.
- van Gasteren, Netty; Feijen, Wim (1995). "[The Binary Search Revisited](#)"  (PDF). AvG127/WF214. (investigates the foundations of the binary search, debunking the myth that it applies only to sorted arrays)

External links [edit]

- [NIST Dictionary of Algorithms and Data Structures: binary search](#) 
- [Binary search implemented in 12 languages](#) 
- [Binary search casual examples - dictionary, array and monotonic function](#) 
- [Benchmark of 7 binary search variants implemented in C](#) 



The Wikibook *Algorithm implementation* has a page on the topic of: [Binary search](#)

Categories: [Search algorithms](#)

This page was last modified on 31 August 2015, at 06:23.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

