WikiJournal of Science, 2019, 2(1):5
doi: 10.15347/wjs/2019.005
Encyclopedic Review Article

Binary search algorithm

Anthony Lin[1]* *et al.*

Abstract

In In computer science, **binary search**, also known as **half-interval search**,[1] **logarithmic search**,[2] or **binary chop**,[3] is a search algorithm that finds a position of a target value within a sorted array.[4] Binary search compares the target value to an element in the middle of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Binary search runs in logarithmic time in the worst case, making $O(\log n)$ comparisons, where n is the number of elements in the array, the O is 'Big O' notation, and log is the logarithm.[5] Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search. There are specialized data structures designed for fast searching, such as hash tables, that can be searched more efficiently than binary search. However, binary search can be used to solve a wider range of problems, such as finding the next-smallest or next-largest element in the array relative to the target even if it is absent from the array.

There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays. Fractional cascading efficiently solves a number of search problems in computational geometry and in numerous other fields. Exponential search extends binary search to unbounded lists. The binary search tree and B-tree data structures are based on binary search.

Algorithm

Binary search works on sorted arrays. Binary search begins by comparing an element in the middle of the array with the target value. If the target value matches the element, its position in the array is returned. If the target value is less than the element, the search continues in the lower half of the array. If the target value is greater than the element, the search continues in the upper half of the array. By doing this, the algorithm eliminates the half in which the target value cannot lie in each iteration.[6]

Procedure

Given an array A of n elements with values or records $A_0, A_1, A_2, \ldots, A_{n-1}$ sorted such that $A_0 \leq A_1 \leq A_2 \leq \cdots \leq A_{n-1}$, and target value T, the following subroutine uses binary search to find the index of T in A.[6]

1. Set L to 0 and R to $n-1$.
2. While $L \leq R$,
    1. Set m (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
    2. If $A_m < T$, set L to $m+1$.
    3. If $A_m > T$, set R to $m-1$.
    4. Else, $A_m = T$; return m.
3. If the search has not returned a value by the time While $L > R$, the search terminates as unsuccessful.

This iterative procedure keeps track of the search boundaries with the two variables L and R. The procedure may be expressed in pseudocode as follows, where the variable names and types remain the same as above, `floor` is the floor function, and `unsuccessful` refers to a specific value that conveys the failure of the search.[6]

***Author correspondence**: by online form
ORCID: [0000-0000-0000-0001]

Received 29-10-2018; accepted 02-07-2019

WikiJournal of Science, 2019, 2(1):5
doi: 10.15347/wjs/2019.005
Encyclopedic Review Article

```
function binary_search(A, n, T):
    L := 0
    R := n - 1
    while L <= R:
        m := floor((L + R) / 2)
        if A[m] < T:
            L := m + 1
        else if A[m] > T:
            R := m - 1
        else:
            return m
    return unsuccessful
```

Alternatively, the algorithm may take the ceiling of $\frac{L+R}{2}$, or the least integer greater than or equal to $\frac{L+R}{2}$. This may change the result if the target value appears more than once in the array.

Alternative procedure

In the above procedure, the algorithm checks whether the middle element (m) is equal to the target (T) in every iteration. Some implementations leave out this check during each iteration. The algorithm would perform this check only when one element is left (when $L = R$). This results in a faster comparison loop, as one comparison is eliminated per iteration. However, it requires one more iteration on average.[1]

Hermann Bottenbruch published the first implementation to leave out this check in 1962.[1][8]

1. Set L to 0 and R to $n - 1$.
2. While $L \neq R$,
    1. Set m (the position of the middle element) to the ceiling of $\frac{L+R}{2}$, which is the least integer greater than or equal to $\frac{L+R}{2}$.
    2. If $A_m > T$, set R to $m - 1$.
    3. Else $A_m \leq T$, set L to m.
3. Now $L = R$, the search is done. If $A_L = T$, return L. Otherwise, the search terminates as unsuccessful.

Where `ceil` is the ceiling function, the pseudocode for this version is:

```
function binary_search_alternative(A, n, T):
    L := 0
    R := n - 1
    while L != R:
        m := ceil((L + R) / 2)
        if A[m] > T:
            R := m - 1
        else:
            L := m
    if A[L] == T:
        return L
    return unsuccessful
```

Duplicate elements

The procedure may return any index whose element is equal to the target value, even if there are duplicate elements in the array. For example, if the array to be searched was $[1,2,3,4,4,5,6,7]$ and the target was 4, then it would be correct for the algorithm to either return the 4th (index 3) or 5th (index 4) element. The regular procedure would return the 4th element (index 3). However, it is sometimes necessary to find the leftmost element or the rightmost element for a target value that is duplicated in the array. In the above example, the 4th element is the leftmost element of the value 4, while the 5th element is the rightmost element of the value 4. The alternative procedure above will always return the index of the rightmost element if such an element exists.[8]

Procedure for finding the leftmost element

To find the leftmost element, the following procedure can be used:[9]

1. Set L to 0 and R to n.
2. While $L < R$,
    1. Set m (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
    2. If $A_m < T$, set L to $m + 1$.
    3. Else $A_m \geq T$, set R to m.
3. Return L.

If $L < n$ and $A_L = T$, then A_L is the leftmost element that equals T. Even if T is not in the array, L is the rank of T in the array, or the number of elements in the array that are less than T.

Where `floor` is the floor function, the pseudocode for this version is:

```
function binary_search_rightmost(A, n, T):
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] > T:
            R := m
        else:
            L := m + 1
    return L - 1
```

Procedure for finding the rightmost element

To find the rightmost element, the following procedure can be used:[9]

1. Set L to 0 and R to n.
2. While $L < R$,
    1. Set m (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
    2. If $A_m > T$, set R to m.
    3. Else $A_m \leq T$, set L to $m + 1$.
3. Return $L - 1$.

If $L > 0$ and $A_{L-1} = T$, then A_{L-1} is the rightmost element that equals T. Even if T is not in the array, $(n - 1) - L$ is the number of elements in the array that are greater than T.

Where `floor` is the floor function, the pseudocode for this version is:

function binary_search_rightmost(A, n, T):

```
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] > T:
            R := m
        else:
            L := m + 1
    return L - 1
```

Approximate matches

The above procedure only performs *exact* matches, finding the position of a target value. However, it is trivial to extend binary search to perform approximate matches because binary search operates on sorted arrays. For example, binary search can be used to compute, for a given 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.[10]

- Rank queries can be performed with the procedure for finding the leftmost element. The number of elements *less than* the target value is returned by the procedure.[10]
- Predecessor queries can be performed with rank queries. If the rank of the target value is r, its predecessor is $r - 1$.[11]
- For successor queries, the procedure for finding the rightmost element can be used. If the result of running the procedure for the target value is r, then the successor of the target value is $r + 1$.[11]

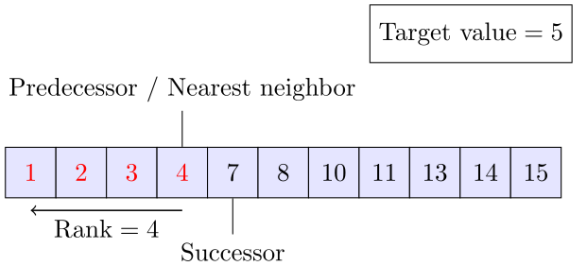

Figure 1 | Binary search can be adapted to compute approximate matches. In the example above, the rank, predecessor, successor, and nearest neighbor are shown for the target value 5, which is not in the array.

- The nearest neighbor of the target value is either its predecessor or successor, whichever is closer.
- Range queries are also straightforward.[11] 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 entries matching those endpoints.[12]

Performance

Number of comparisons

In terms of the number of comparisons, the performance of binary search can be analyzed by viewing the run of the procedure on a binary tree. The root node of the tree is the middle element of the array. The middle element of the lower half is the left child node of the root, and the middle element of the upper half is the right child node of the root. The rest of the tree is built in a similar fashion. 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.[5][13]

In the worst case, binary search makes $\lfloor \log_2(n) + 1 \rfloor$ iterations of the comparison loop, where the $\lfloor \ \rfloor$ notation denotes the floor function that yields the greatest integer less than or equal to the argument, and $\log_2$ is the binary logarithm. This is because the worst case is reached when the search reaches the deepest level of the tree, and there are always $\lfloor \log_2(n) + 1 \rfloor$ levels in the tree for any binary search.

The worst case may also be reached when the target element is not in the array. If n is one less than a power of

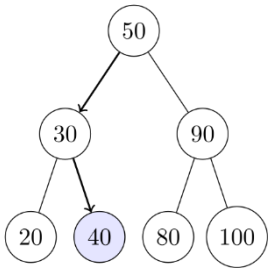

Figure 2 | A tree representing binary search. The array being searched here is [20, 30, 40, 50, 80, 90, 100], and the target value is 40.

two, then this is always the case. Otherwise, the search may perform $\lfloor \log_2(n) + 1 \rfloor$ iterations if the search reaches the deepest level of the tree. However, it may make $\lfloor \log_2(n) \rfloor$ iterations, which is one less than the worst case, if the search ends at the second-deepest level of the tree.[14]

On average, assuming that each element is equally likely to be searched, binary search makes $\lfloor \log_2(n) \rfloor + 1 - \left(2^{\lfloor \log_2(n) \rfloor + 1} - \lfloor \log_2(n) \rfloor - 2\right)/n$ iterations when the target element is in the array. This is approximately equal to $\log_2(n) - 1$ iterations. When the target element is not in the array, binary search makes $\lfloor \log_2(n) \rfloor + 2 - 2^{\lfloor \log_2(n) \rfloor + 1}/(n + 1)$ iterations on average, assuming that the range between and outside elements is equally likely to be searched.[13]

In the best case, where the target value is the middle element of the array, its position is returned after one iteration.[15]

In terms of iterations, no search algorithm that works only by comparing elements can exhibit better average and worst-case performance than binary search. The comparison tree representing binary search has the fewest levels possible as every level above the lowest level of the tree is filled completely.[a] Otherwise, the search algorithm can eliminate few elements in an iteration, increasing the number of iterations required in the average and worst case. This is the case for other search algorithms based on comparisons, as while they may work faster on some target values, the average performance over *all* elements is worse than binary search. By dividing the array in half, binary search ensures that the size of both subarrays are as similar as possible.[13]

Space complexity

Binary search requires three pointers to elements, which may be array indices or pointers to memory locations, regardless of the size of the array. However, it requires at least $\lceil \log_2(n) \rceil$ bits to encode a pointer to an element of an array with n elements.[16] Therefore, the space complexity of binary search is $O(\log n)$. In addition, it takes $O(n)$ space to store the array.

Derivation of average case

The average number of iterations performed by binary search depends on the probability of each element being searched. The average case is different for successful searches and unsuccessful searches. It will be assumed that each element is equally likely to be searched for successful searches. For unsuccessful searches, it will be assumed that the intervals between and outside elements are equally likely to be searched. The average case for successful searches is the number of iterations required to search every element exactly once, divided by n, the number of elements. The average case for unsuccessful searches is the number of iterations required to search an element within every interval exactly once, divided by the $n + 1$ intervals.[13]

Successful searches

In the binary tree representation, a successful search can be represented by a path from the root to the target node, called an *internal path*. The length of a path is the number of edges (connections between nodes) that the path passes through. The number of iterations performed by a search, given that the corresponding path has length l , is $l + 1$ counting the initial iteration. The *internal path length* is the sum of the lengths of all

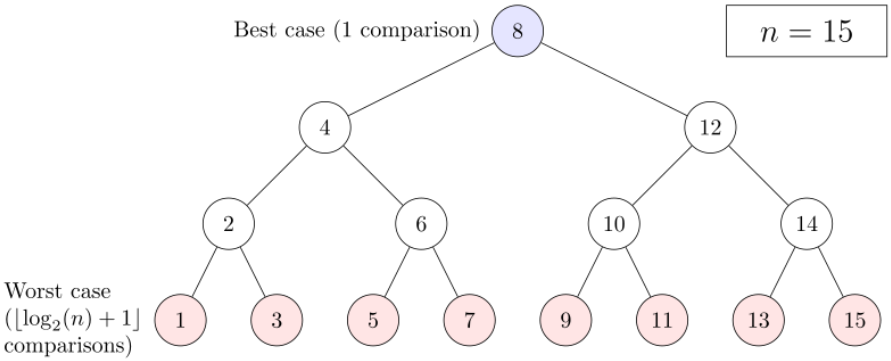

Figure 3 | The worst case is reached when the search reaches the deepest level of the tree, while the best case is reached when the target value is the middle element.

unique internal paths. Since there is only one path from the root to any single node, each internal path represents a search for a specific element. If there are n elements, which is a positive integer, and the internal path length is $I(n)$ then the average number of iterations for a successful search $T(n) = 1 + \frac{I(n)}{n}$, with the one iteration added to count the initial iteration.[13]

Since binary search is the optimal algorithm for searching with comparisons, this problem is reduced to calculating the minimum internal path length of all binary trees with n nodes, which is equal to:[17]

$$I(n) = \sum_{k=1}^{n} \lfloor log_2(k)$$

For example, in a 7-element array, the root requires one iteration, the two elements below the root require two iterations, and the four elements below require three iterations. In this case, the internal path length is:[17]

$$\begin{aligned}\sum_{k=1}^{7} \lfloor log_2(k)\rfloor &= 0 + 2(1) + 4(2) \\ &= 2 + 8 \\ &= 10\end{aligned}$$

The average number of iterations would be $1 + \frac{10}{7} = 2\frac{3}{7}$ based on the equation for the average case. The sum for $I(n)$ can be simplified to:[13]

$$\begin{aligned}I(n) &= \sum_{k=1}^{n} \lfloor log_2(k)\rfloor \\ &= (n+1)\lfloor log_2(n+1)\rfloor - 2^{\lfloor log_2(n+1)\rfloor+1} + 2\end{aligned}$$

Substituting the equation for $I(n)$ into the equation for $T(n)$:[13]

$$\begin{aligned}T(n) &= 1 + \frac{(n+1)\lfloor log_2(n+1)\rfloor - 2^{\lfloor log_2(n+1)\rfloor+1} + 2}{n} \\ &= \lfloor log_2(n)\rfloor + 1 - \left(2^{\lfloor log_2(n)\rfloor+1} - \lfloor log_2(n)\rfloor - 2\right)/n\end{aligned}$$

For integer n, this is equivalent to the equation for the average case on a successful search specified above.

Unsuccessful searches

Unsuccessful searches can be represented by augmenting the tree with *external nodes*, which forms an *extended binary tree*. If an internal node, or a node present in the tree, has fewer than two child nodes, then additional child nodes, called external nodes, are added so that each internal node has two children. By doing so, an unsuccessful search can be represented as a path to an external node, whose parent is the single element that remains during the last iteration. An *external path* is a path from the root to an external node. The *external path length* is the sum of the lengths of all unique external paths. If there are n elements, which is a positive integer, and the external path length is $E(n)$, then the average number of iterations for an unsuccessful search $T'(n) = \frac{E(n)}{n+1}$, with the one iteration added to count the initial iteration. The external path length is divided by $n + 1$ instead of n because there are $n + 1$ external paths, representing the intervals between and outside the elements of the array.[13]

This problem can similarly be reduced to determining the minimum external path length of all binary trees with n nodes. For all binary trees, the external path length is equal to the internal path length plus $2n$.[17] Substituting the equation for $I(n)$:[13]

$$\begin{aligned}E(n) &= I(n) + 2n \\ &= \left[(n+1)\lfloor log_2(n+1)\rfloor - 2^{\lfloor log_2(n+1)\rfloor+1} + 2\right] + 2n \\ &= (n+1)(\lfloor log_2(n)\rfloor + 2) - 2^{\lfloor log_2(n)\rfloor+1}\end{aligned}$$

Substituting the equation for $E(n)$ into the equation for $T'(n)$, the average case for unsuccessful searches can be determined:[13]

$$\begin{aligned}T'(n) &= \frac{(n+1)(\lfloor log_2(n)\rfloor + 2) - 2^{\lfloor log_2(n)\rfloor+1}}{(n+1)} \\ &= \lfloor log_2(n)\rfloor + 2 - 2^{\lfloor log_2(n)\rfloor+1}/(n+1)\end{aligned}$$

Performance of alternative procedure

Each iteration of the binary search procedure defined above makes one or two comparisons, checking if the middle element is equal to the target in each iteration. Assuming that each element is equally likely to be searched, each iteration makes 1.5 comparisons on average. A variation of the algorithm checks whether the middle element is equal to the target at the end of the search. On average, this eliminates half a comparison from each iteration. This slightly cuts the time taken per iteration on most computers. However, it guarantees that the search takes the maximum number of iterations, on average adding one iteration to the search. Because the comparison loop is performed only $\lfloor \log_2(n) + 1 \rfloor$ times in the worst case, the slight increase in efficiency per iteration does not compensate for the extra iteration for all but very large n.[b][18][19]

Running time and cache use

In analyzing the performance of binary search, another consideration is the time required to compare two elements. For integers and strings, the time required increases linearly as the encoding length (usually the

number of bits) of the elements increase. For example, comparing a pair of 64-bit unsigned integers would require comparing up to double the bits as comparing a pair of 32-bit unsigned integers. The worst case is achieved when the integers are equal. This can be significant when the encoding lengths of the elements are large, such as with large integer types or long strings, which makes comparing elements expensive. Furthermore, comparing floating-point values (the most common digital representation of real numbers) is often more expensive than comparing integers or short strings.

On most computer architectures, the processor has a hardware cache separate from RAM. Since they are located within the processor itself, caches are much faster to access but usually store much less data than RAM. Therefore, most processors store memory locations that have been accessed recently, along with memory locations close to it. For example, when an array element is accessed, the element itself may be stored along with the elements that are stored close to it in RAM, making it faster to sequentially access array elements that are close in index to each other (locality of reference). On a sorted array, binary search can jump to distant memory locations if the array is large, unlike algorithms (such as linear search and linear probing in hash tables) which access elements in sequence. This adds slightly to the running time of binary search for large arrays on most systems.[20]

Binary search versus other schemes

Sorted arrays with binary search are a very inefficient solution when insertion and deletion operations are interleaved with retrieval, taking $O(n)$ time for each such operation. In addition, sorted arrays can complicate memory use especially when elements are often inserted into the array.[21] There are other data structures that support much more efficient insertion and deletion. Binary search can be used to perform exact matching and set membership (determining whether a target value is in a collection of values). There are data structures that support faster exact matching and set membership. However, unlike many other searching schemes, binary search can be used for efficient approximate matching, usually performing such matches in $O(\log n)$ time regardless of the type or structure of the values themselves.[22] In addition, there are some operations, like finding the smallest and largest element, that can be performed efficiently on a sorted array.[10]

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, although the array needs to be sorted beforehand.[c][24] All sorting algorithms based on comparing elements, such as quicksort and merge sort, require at least $O(n \log n)$ comparisons in the worst case.[25] Unlike linear search, binary search can be used for efficient approximate matching. There are operations such as finding the smallest and largest element that can be done efficiently on a sorted array but not on an unsorted array.[26]

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, although the array needs to be sorted beforehand.[lower-alpha 3][24] All sorting algorithms based on comparing elements, such as quicksort and merge sort, require at least $O(n \log n)$ comparisons in the worst case.[25] Unlike linear search, binary search can be used for efficient approximate matching. There are operations such as finding the smallest and largest element that can be done efficiently on a sorted array but not on an unsorted array.[26]

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 each record in the tree can be searched using an algorithm similar to binary search, taking on average logarithmic time. Insertion and deletion also require on average logarithmic time in binary search trees. This can be 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, including range and approximate queries.[22][27]

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 even applies to balanced binary search trees, binary search trees that balance their own nodes, because they rarely produce the tree with the fewest possible levels. Except for balanced

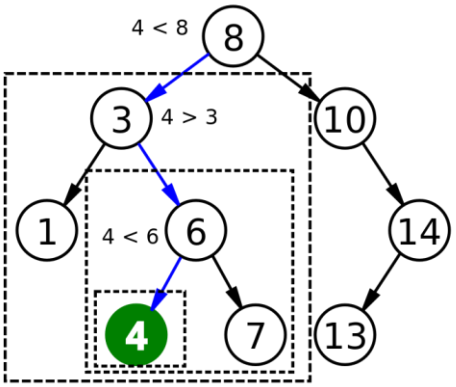

Figure 4 | Binary search trees are searched using an algorithm similar to binary search.
Chris Martin, public domain

binary search trees, the tree may be severely imbalanced with few internal nodes with two children, resulting in the average and worst-case search time approaching n comparisons.[d] Binary search trees take more space than sorted arrays.[29]

Binary search trees lend themselves to fast searching in external memory stored in hard disks, as binary search trees can be efficiently structured in filesystems. The B-tree generalizes this method of tree organization. B-trees are frequently used to organize long-term storage such as databases and filesystems.[30][31]

Hashing

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.[32] Most hash table implementations require only amortized constant time on average.[e][34] 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.[35] Binary search is ideal for such matches, performing them in logarithmic time. Binary search also supports approximate matches. Some operations, like finding the smallest and largest element, can be done efficiently on sorted arrays but not on hash tables.[22]

Set membership algorithms

A related problem to search is set membership. Any algorithm that does lookup, like binary search, can also be used for set membership. There are other algorithms that are more specifically suited for set membership. A bit array is the simplest, useful when the range of keys is limited. It compactly stores a collection of bits, with each bit representing a single key within the range of keys. Bit arrays are very fast, requiring only $O(1)$ time.[36] The Judy1 type of Judy array handles 64-bit keys efficiently.[37]

For approximate results, Bloom filters, another probabilistic data structure based on hashing, store a set of keys by encoding the keys using a bit array and multiple hash functions. Bloom filters are much more space-efficient than bit arrays in most cases and not much slower: k hash functions, membership queries require only $O(k)$ time. However, Bloom filters suffer from false positives.[f][g][39]

Other data structures

There exist data structures that may improve on 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. These specialized data structures are usually only faster because they take advantage of 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 lack that attribute.[22] As long as the keys can be ordered, these operations can always be done at least efficiently on a sorted array regardless of the keys. Some structures, such as Judy arrays, use a combination of approaches to mitigate this while retaining efficiency and the ability to perform approximate matching.[37]

Variations

Uniform binary search

Main article: Uniform binary search

Uniform binary search stores, instead of the lower and upper bounds, the difference in the index of the middle element from the current iteration to the next iteration. A lookup tablecontaining the differences is computed beforehand. For example, if the array to be searched is

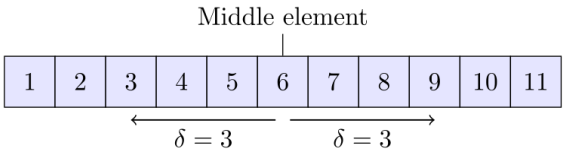

Figure 5 | Uniform binary search stores the difference between the current and the two next possible middle elements instead of specific bounds.

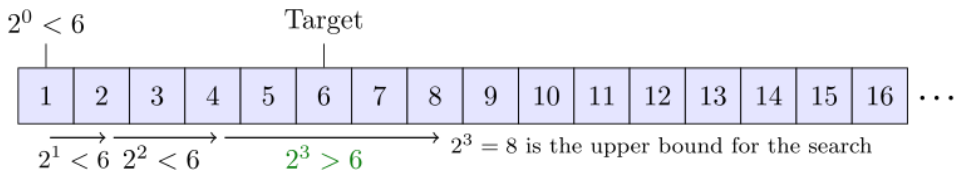

Figure 6 | Visualization of exponential searching finding the upper bound for the subsequent binary search.

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], the middle element (m) would be 6. In this case, the middle element of the left subarray ([1, 2, 3, 4, 5]) is 3 and the middle element of the right subarray ([7, 8, 9, 10, 11]) is 9. Uniform binary search would store the value of 3 as both indices differ from 6 by this same amount.[40] To reduce the search space, the algorithm either adds or subtracts this change from the index of the middle element. Uniform binary search may be faster on systems where it is inefficient to calculate the midpoint, such as on decimal computers.[41]

Exponential search

Main article: Exponential search

Exponential search extends binary search to unbounded lists. 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 $\lfloor \log_2 x + 1 \rfloor$ iterations before binary search is started and at most $\lfloor \log_2 x \rfloor$ iterations of the binary search, where x is the position of the target value. Exponential search works on bounded lists, but becomes an improvement over binary search only if the target value lies near the beginning of the array.[42]

Interpolation search

Main article: Interpolation search

Instead of calculating the midpoint, interpolation search estimates the position of the target value, taking into account the lowest and highest elements in the array as well as length of the array. 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.[43]

A common interpolation function is linear interpolation. If A is the array, L, R are the lower and upper bounds respectively, and T is the target, then the target is estimated to be about $(T - A_L)/(A_R - A_L)$ of the way between L and R. When linear interpolation is used, and the distribution of the array elements is uniform or near uniform, interpolation search makes $O(\log \log n)$ comparisons.[43][44][45]

In practice, interpolation search is slower than binary search for small arrays, as interpolation search requires extra computation. Its time complexity grows more slowly than binary search, but this only compensates for the extra computation for large arrays.[43]

Fractional cascading

Main article: Fractional cascading

Fractional cascading is a technique that speeds up binary searches for the same element in multiple sorted arrays. Searching each array separately requires $O(k \log n)$ time, where k is the number of arrays. Fractional cascading reduces this to $O(k + \log n)$ by storing specific information in each array about each element and its position in the other arrays.[46][47]

Fractional cascading was originally developed to efficiently solve various computational geometry problems. Fractional cascading has been applied elsewhere, such as in data mining and Internet Protocol routing.[46]

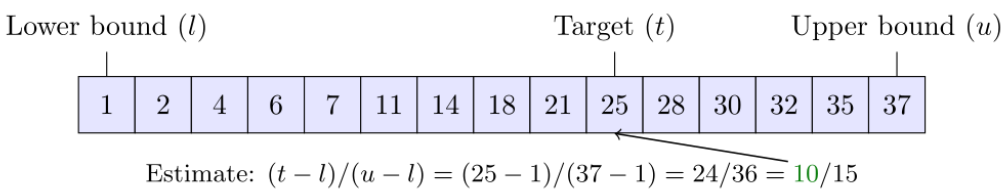

Figure 7 | Visualization of interpolation search. In this case, no searching is needed because the estimate of the target's location within the array is correct. Other implementations may specify another function for estimating the target's location.

WikiJournal of Science, 2019, 2(1):5
doi: 10.15347/wjs/2019.005
Encyclopedic Review Article

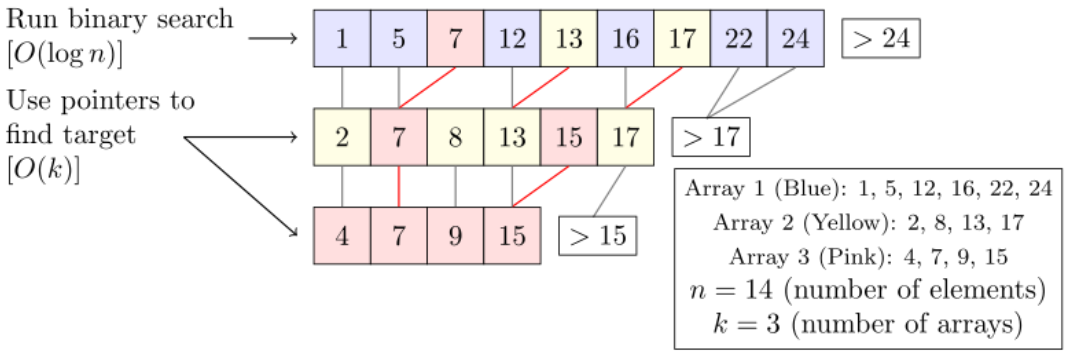

Figure 8 | In fractional cascading, each array has pointers to every second element of another array, so only one binary search has to be performed to search all the arrays.

Generalization to graphs

Binary search has been generalized to work on certain types of graphs, where the target value is stored in a vertex instead of an array element. Binary search trees are one such generalization—when a vertex (node) in the tree is queried, the algorithm either learns that the vertex is the target, or otherwise which subtree the target would be located in. However, this can be further generalized as follows: given an undirected, positively weighted graph and a target vertex, the algorithm learns upon querying a vertex that it is equal to the target, or it is given an incident edge that is on the shortest path from the queried vertex to the target. The standard binary search algorithm is simply the case where the graph is a path. Similarly, binary search trees are the case where the edges to the left or right subtrees are given when the queried vertex is unequal to the target. For all undirected, positively weighted graphs, there is an algorithm that finds the target vertex in $O(\log n)$ queries in the worst case.[48]

Noisy binary search

Noisy binary search algorithms solve the case where the algorithm cannot reliably compare elements of the array. For each pair of elements, there is a certain probability that the algorithm makes the wrong comparison. Noisy binary search can find the correct position of the target with a given probability that controls the reliability of the yielded position. Every noisy binary search procedure must make at least $(1-\tau)\frac{\log_2(n)}{H(p)} - \frac{10}{H(p)}$ comparisons on average, where $(1-\tau)\frac{\log_2(n)}{H(p)} - \frac{10}{H(p)}$ is the binary entropy function and τ is the probability that the procedure yields the wrong position.[49][50][51] The noisy binary search problem can be considered as a case of the Rényi-Ulam game,[52] a variant of Twenty Questions where the answers may be wrong.[53]

Quantum binary search

Classical computers are bounded to the worst case of exactly $\lfloor \log_2 n + 1 \rfloor$ iterations when performing binary search. Quantum algorithms for binary search are still bounded to a proportion of $\log_2 n$ queries (representing iterations of the classical procedure), but the constant factor is less than one, providing for a lower time complexity on quantum computers. Any *exact* quantum binary search procedure—that is, a procedure that always yields the correct result—requires at least $\frac{1}{\pi}(\ln n - 1) \approx 0.22 \log_2 n$ queries in the worst case, where ln is the natural logarithm.[54] There is an exact quantum binary search procedure that runs in $4 \log_{605} n \approx 0.433 \log_2 n$ queries in the worst case.[55] In comparison, Grover's algorithm is the optimal quantum algorithm for searching an unordered list of elements, and it requires $O(\sqrt{n})$ queries.[56]

History

The idea of sorting a list of items to allow for faster searching dates back to antiquity. The earliest known example was the Inakibit-Anu tablet from Babylon dating back to c. 200 BCE. The tablet contained about 500 sexagesimalnumbers and their reciprocals sorted in lexicographical order, which made searching for a specific entry easier. In addition, several lists of names that were sorted by their first letter were discovered on the Aegean Islands. Catholicon, a Latin dictionary finished in 1286 CE, was the first work to describe rules for sorting words into alphabetical order, as opposed to just the first few letters.[8]

In 1946, John Mauchly made the first mention of binary search as part of the Moore School Lectures, a seminal and foundational college course in computing.[8] In 1957, William Wesley Peterson published the first

method for interpolation search.[8][57] Every published binary search algorithm worked only for arrays whose length is one less than a power of two[h] until 1960, when Derrick Henry Lehmer published a binary search algorithm that worked on all arrays.[59] 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.[7] The uniform binary search was developed by A. K. Chandra of Stanford University in 1971.[8] In 1986, Bernard Chazelle and Leonidas J. Guibas introduced fractional cascading as a method to solve numerous search problems in computational geometry.[46][60][61]

Implementation issues

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky ... — Donald Knuth[2]

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, mainly because the incorrect implementations failed to run or returned a wrong answer in rare edge cases.[62] A study published in 1988 shows that accurate code for it is only found in five out of twenty textbooks.[63] 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.[64]

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 $\frac{L+R}{2}$, 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 $L + \frac{R-L}{2}$.[65]

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 most of the programmers who incorrectly implemented binary search made an error in defining the exit conditions.[7][66]

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.[67]
- C++'s Standard Template Library provides the functions `binary_search()`, `lower_bound()`, `upper_bound()` and `equal_range()`.[68]
- COBOL provides the `SEARCH ALL` verb for performing binary searches on COBOL ordered tables.[69]
- 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.[70]
- Java offers a set of overloaded `binarySearch()` static methods in the classes `Arrays` and `Collectionsin` the standard `java.`util package for performing binary searches on Java arrays and on `Lists`, respectively.[71][72]
- 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)`.[73]
- For Objective-C, the Cocoa framework provides the `NSArray -indexOfObject:inSortedRange:options:usingComparator:` method in Mac OS X 10.6+.[74] Apple's Core Foundation C framework also contains a `CFArrayBSearchValues()` function.[75]
- Python provides the `bisect` module.[76]
- Ruby's Array class includes a `bsearch` method with built-in approximate matching.[77]

Additional information

Acknowledgements

I would like to thank all the Wikipedia editors who have contributed to the article, as well as the Wikipedia editors who have contributed to the good article review and featured article review of the article.

Notes

a. Any search algorithm based solely on comparisons can be represented using a binary comparison tree. An *internal path* is any path from the root to an existing node. Let l be the *internal path length*, the sum of the lengths of all internal paths. If each element is equally likely to be searched, the average case is $1 + \frac{I}{n}$ or simply one plus the average of all the internal path lengths of the tree. This is because internal paths represent the elements that the search algorithm compares to the target. The lengths of these internal paths represent the number of iterations *after* the root node. Adding the average of these lengths to the one iteration at the root yields the average case. Therefore, to minimize the average number of comparisons, the internal path length l must be minimized. It turns out that the tree for binary search minimizes the internal path length. Knuth 1998 proved that the *external path* length (the path length over all nodes where both children are present for each already-existing node) is minimized when the external nodes (the nodes with no children) lie within two consecutive levels of the tree. This also applies to internal paths as internal path length l is linearly related to external path length . For any tree of n nodes, $I = E - 2n$. When each subtree has a similar number of nodes, or equivalently the array is divided into halves in each iteration, the external nodes as well as their interior parent nodes lie within two levels. It follows that binary search minimizes the number of average comparisons as its comparison tree has the lowest possible internal path length.[13]

b. Knuth 1998 showed on his MIX computer model, which Knuth designed as a representation of an ordinary computer, that the average running time of this variation for a successful search is $17.5 \log_2 n + 17$ units of time compared to $18 \log_2 n - 16$ units for regular binary search. The time complexity for this variation grows slightly more slowly, but at the cost of higher initial complexity. [18]

c. Knuth 1998 performed a formal time performance analysis of both of these search algorithms. On Knuth's MIX computer, which Knuth designed as a representation of an ordinary computer, binary search takes on average $18 \log n - 16$ units of time for a successful search, while linear search with a sentinel node at the end of the list takes $1.75n + 8.5 - \frac{n \bmod 2}{4n}$ units. Linear search has lower initial complexity because it requires minimal computation, but it quickly outgrows binary search in complexity. On the MIX computer, binary search only outperforms linear search with a sentinel if $n > 44$.[13][23]

d. Inserting the values in sorted order or in an alternating lowest-highest key pattern will result in a binary search tree that maximizes the average and worst-case search time.[28]

e. It is possible to search some hash table implementations in guaranteed constant time.[33]

f. This is because simply setting all of the bits which the hash functions point to for a specific key can affect queries for other keys which have a common hash location for one or more of the functions.[38]

g. There exist improvements of the Bloom filter which improve on its complexity or support deletion; for example, the cuckoo filter exploits cuckoo hashing to gain these advantages.[38]

h. That is, arrays of length 1, 3, 7, 15, 31 ...[58]

Citations

1. *Willams, Jr., Louis F. (22 April 1976). A modification to the half-interval search (binary search) method. Proceedings of the 14th ACM Southeast Conference. ACM. pp. 95–101. doi:10.1145/503561.503582. Retrieved 29 June 2018.*
2. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Binary search".
3. Butterfield & Ngondi 2016, p. 46.
4. Cormen et al. 2009, p. 39.
5. Flores, Ivan; Madpis, George (1 September 1971). "Average binary search length for dense ordered lists". *Communications of the ACM* **14** (9): 602–603. doi:10.1145/362663.362752. ISSN 0001-0782. Retrieved 29 June 2018.
6. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Algorithm B".
7. Bottenbruch, Hermann (1 April 1962). "Structure and use of ALGOL 60". *Journal of the ACM* **9** (2): 161–221. doi:10.1145/321119.321120. ISSN 0004-5411. Retrieved 30 June 2018. Procedure is described at p. 214 (§43), titled "Program for Binary Search".
8. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "History and bibliography".
9. Kasahara & Morishita 2006, pp. 8–9.
10. Sedgewick & Wayne 2011, §3.1, subsection "Rank and selection".
11. Goldman & Goldman 2008, pp. 461–463.
12. Sedgewick & Wayne 2011, §3.1, subsection "Range queries".
13. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Further analysis of binary search".
14. Knuth 1998, §6.2.1 ("Searching an ordered table"), "Theorem B".
15. Chang 2003, p. 169.

16. Shannon, Claude E. (July 1948). "A Mathematical Theory of Communication". *Bell System Technical Journal* **27** (3): 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x.
17. Knuth 1997, §2.3.4.5 ("Path length").
18. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Exercise 23".
19. Rolfe, Timothy J. (1997). "Analytic derivation of comparisons in binary search". *ACM SIGNUM Newsletter* **32** (4): 15–19. doi:10.1145/289251.289255.
20. Khuong, Paul-Virak; Morin, Pat. "Array Layouts for Comparison-Based Searching". *Journal of Experimental Algorithmics* (Article 1.3) **22**. doi:10.1145/289251.289255.
21. Knuth 1997, §2.2.2 ("Sequential Allocation").
22. Beame, Paul; Fich, Faith E. (2001). "Optimal bounds for the predecessor problem and related problems". *Journal of Computer and System Sciences* **65** (1): 38–72. doi:10.1006/jcss.2002.1822.
23. Knuth 1998, Answers to Exercises (§6.2.1) for "Exercise 5".
24. Knuth 1998, §6.2.1 ("Searching an ordered table").
25. Knuth 1998, §5.3.1 ("Minimum-Comparison sorting").
26. Sedgewick & Wayne 2011, §3.2 ("Ordered symbol tables").
27. Sedgewick & Wayne 2011, §3.2 ("Binary Search Trees"), subsection "Order-based methods and deletion".
28. Knuth 1998, §6.2.2 ("Binary tree searching"), subsection "But what about the worst case?".
29. Sedgewick & Wayne 2011, §3.5 ("Applications"), "Which symbol-table implementation should I use?".
30. Knuth 1998, §5.4.9 ("Disks and Drums").
31. Knuth 1998, §6.2.4 ("Multiway trees").
32. Knuth 1998, §6.4 ("Hashing").
33. Knuth 1998, §6.4 ("Hashing"), subsection "History".
34. Dietzfelbinger, Martin; Karlin, Anna; Mehlhorn, Kurt; Meyer auf der Heide, Friedhelm; Rohnert, Hans; Tarjan, Robert E.(August 1994). "Dynamic perfect hashing: upper and lower bounds". *SIAM Journal on Computing* **23** (4): 738–761. doi:10.1137/S0097539791194094.
35. *Morin, Pat. "Hash tables" (PDF). p. 1. Retrieved 28 March 2016.*
36. Knuth 2011, §7.1.3 ("Bitwise Tricks and Techniques").
37. Silverstein, Alan, *Judy IV shop manual*, Hewlett-Packard
38. *Fan, Bin; Andersen, Dave G.; Kaminsky, Michael; Mitzenmacher, Michael D. (2014). Cuckoo filter: practically better than Bloom. Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. pp. 75–88. doi:10.1145/2674005.2674994.*
39. Bloom, Burton H. (1970). "Space/time trade-offs in hash coding with allowable errors". *Communications of the ACM* **13**(7): 422–426. doi:10.1145/362686.362692.
40. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "An important variation".
41. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Algorithm U".
42. Moffat & Turpin 2002, p. 33.
43. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Interpolation search".
44. Knuth 1998, §6.2.1 ("Searching an ordered table"), subsection "Exercise 22".
45. Perl, Yehoshua; Itai, Alon; Avni, Haim (1978). "Interpolation search—a log log *n* search". *Communications of the ACM***21** (7): 550–553. doi:10.1145/359545.359557.
46. *Chazelle, Bernard; Liu, Ding (6 July 2001). Lower bounds for intersection searching and fractional cascading in higher dimension. 33rd ACM Symposium on Theory of Computing. ACM. pp. 322–329. doi:10.1145/380752.380818. ISBN 978-1-58113-349-3. Retrieved 30 June 2018.*
47. Chazelle, Bernard; Liu, Ding (1 March 2004). "Lower bounds for intersection searching and fractional cascading in higher dimension" (in en). *Journal of Computer and System Sciences* **68** (2): 269–284. doi:10.1016/j.jcss.2003.07.003. ISSN 0022-0000. Retrieved 30 June 2018.
48. *Emamjomeh-Zadeh, Ehsan; Kempe, David; Singhal, Vikrant (2016). Deterministic and probabilistic binary search in graphs. 48th ACM Symposium on Theory of Computing. pp. 519–532. arXiv:1503.00805. doi:10.1145/2897518.2897656.*
49. *Ben-Or, Michael; Hassidim, Avinatan (2008). "The Bayesian learner is optimal for noisy binary search (and pretty good for quantum as well)" (PDF). 49th Symposium on Foundations of Computer Science. pp. 221–230. doi:10.1109/FOCS.2008.58. ISBN 978-0-7695-3436-7.*
50. Pelc, Andrzej (1989). "Searching with known error probability". *Theoretical Computer Science* **63** (2): 185–202. doi:10.1016/0304-3975(89)90077-7.
51. *Rivest, Ronald L.; Meyer, Albert R.; Kleitman, Daniel J.; Winklmann, K. Coping with errors in binary search procedures. 10th ACM Symposium on Theory of Computing. doi:10.1145/800133.804351.*
52. Pelc, Andrzej (2002). "Searching games with errors—fifty years of coping with liars". *Theoretical Computer Science***270** (1–2): 71–109. doi:10.1016/S0304-3975(01)00303-6.
53. Rényi, Alfréd (1961). "On a problem in information theory" (in Hungarian). *Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei* **6**: 505–516.
54. Høyer, Peter; Neerbek, Jan; Shi, Yaoyun (2002). "Quantum complexities of ordered searching, sorting, and element distinctness". *Algorithmica* **34** (4): 429–448. doi:10.1007/s00453-002-0976-3.
55. Childs, Andrew M.; Landahl, Andrew J.; Parrilo, Pablo A. (2007). "Quantum algorithms for the ordered search problem via semidefinite programming". *Physical Review A* **75** (3): 032335. doi:10.1103/PhysRevA.75.032335.
56. *Grover, Lov K. (1996). A fast quantum mechanical algorithm for database search. 28th ACM Symposium on Theory of Computing. Philadelphia, PA. pp. 212–219. arXiv:quant-ph/9605043. doi:10.1145/237814.237866.*
57. Peterson, William Wesley (1957). "Addressing for random-access storage". *IBM Journal of Research and Development* **1** (2): 130–146. doi:10.1147/rd.12.0130.
58. "2^n–1". OEIS A000225. Retrieved 7 May 2016.
59. *Lehmer, Derrick (1960). Teaching combinatorial tricks to a computer. Proceedings of Symposia in Applied Mathematics. **10**. pp. 180–181. doi:10.1090/psapm/010.*
60. Chazelle, Bernard; Guibas, Leonidas J. (1986). "Fractional cascading: I. A data structuring technique". *Algorithmica***1** (1): 133–162. doi:10.1007/BF01840440.
61. Chazelle, Bernard; Guibas, Leonidas J. (1986), "Fractional cascading: II. Applications", *Algorithmica* **1** (1), doi:10.1007/BF01840441
62. Bentley 2000, §4.1 ("The Challenge of Binary Search").
63. Pattis, Richard E. (1988). "Textbook errors in binary searching". *SIGCSE Bulletin* **20**: 190–194. doi:10.1145/52965.53012.
64. *Bloch, Joshua (2 June 2006). "Extra, extra – read all about it: nearly all binary searches and mergesorts are broken". Google Research Blog. Retrieved 21 April 2016.*
65. Ruggieri, Salvatore (2003). "On computing the semi-sum of two integers". *Information Processing Letters* **87** (2): 67–71. doi:10.1016/S0020-0190(03)00263-1.
66. Bentley 2000, §4.4 ("Principles").
67. *"bsearch – binary search a sorted table". The Open Group Base Specifications (7th ed.). The Open Group. 2013. Retrieved 28 March 2016.*
68. Stroustrup 2013, p. 945.
69. Unisys (2012), *COBOL ANSI-85 programming reference manual*, **1**
70. *"Package sort". The Go Programming Language. Retrieved 28 April 2016.*
71. *"java.util.Arrays". Java Platform Standard Edition 8 Documentation. Oracle Corporation. Retrieved 1 May 2016.*
72. *"java.util.Collections". Java Platform Standard Edition 8 Documentation. Oracle Corporation. Retrieved 1 May 2016.*
73. *"List<T>.BinarySearch method (T)". Microsoft Developer Network. Retrieved 10 April 2016.*
74. *"NSArray". Mac Developer Library. Apple Inc. Retrieved 1 May 2016.*
75. *"CFArray". Mac Developer Library. Apple Inc. Retrieved 1 May 2016.*
76. *"8.6. bisect — Array bisection algorithm". The Python Standard Library. Python Software Foundation. Retrieved 26 March 2018.*
77. Fitzgerald 2007, p. 152.

Works

- *Bentley, Jon* (2000). Programming pearls (2nd ed.). *Addison-Wesley*. *ISBN 978-0-201-65788-3*.
- Butterfield, Andrew; Ngondi, Gerard E. (2016). A dictionary of computer science (7th ed.). Oxford, UK: *Oxford University Press*. *ISBN 978-0-19-968897-5*.
- Chang, Shi-Kuo (2003). Data structures and algorithms. Software Engineering and Knowledge Engineering. **13**. Singapore: *World Scientific*. *ISBN 978-981-238-348-8*.
- *Cormen, Thomas H.*; *Leiserson, Charles E.*; *Rivest, Ronald L.*; *Stein, Clifford* (2009). *Introduction to algorithms* (3rd ed.). MIT Press and McGraw-Hill. *ISBN 978-0-262-03384-8*.
- Fitzgerald, Michael (2007). Ruby pocket reference. *Sebastopol, California*: *O'Reilly Media*. *ISBN 978-1-4919-2601-7*.

- Goldman, Sally A.; Goldman, Kenneth J. (2008). A practical guide to data structures and algorithms using Java. Boca Raton, Florida: *CRC Press*. *ISBN 978-1-58488-455-2*.
- Kasahara, Masahiro; Morishita, Shinichi (2006). Large-scale genome sequence processing. London, UK: Imperial College Press. *ISBN 978-1-86094-635-6*.
- *Knuth, Donald* (1997). Fundamental algorithms. *The Art of Computer Programming*. **1** (3rd ed.). Reading, MA: Addison-Wesley Professional. *ISBN 978-0-201-89683-1*.
- *Knuth, Donald* (1998). Sorting and searching. *The Art of Computer Programming*. **3** (2nd ed.). Reading, MA: Addison-Wesley Professional. *ISBN 978-0-201-89685-5*.
- *Knuth, Donald* (2011). Combinatorial algorithms. *The Art of Computer Programming*. **4A** (1st ed.). Reading, MA: Addison-Wesley Professional. *ISBN 978-0-201-03804-0*.
- Moffat, Alistair; Turpin, Andrew (2002). Compression and coding algorithms. Hamburg, Germany: Kluwer Academic Publishers. *doi:10.1007/978-1-4615-0935-6*. *ISBN 978-0-7923-7668-2*.
- *Sedgewick, Robert*; Wayne, Kevin (2011). Algorithms (4th ed.). Upper Saddle River, New Jersey: Addison-Wesley Professional. *ISBN 978-0-321-57351-3*.
- *Stroustrup, Bjarne* (2013). The C++ programming language (4th ed.). Upper Saddle River, New Jersey: Addison-Wesley Professional. *ISBN 978-0-321-56384-2*.